Theoretical Computer Science 525 (2014) 42–54

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Generalized substring compression

Orgad Keller [a], Tsvi Kopelowitz [a], Shir Landau Feibish [b,*,1], Moshe Lewenstein [a,2]

[a] Department of Computer Science, Bar-Ilan University, Ramat-Gan, Israel
[b] Blavatnik School of Computer Sciences, Tel-Aviv University, Israel

ARTICLE INFO

Keywords:
Data compression
Lempel–Ziv compression
Suffix tree
Range searching

ABSTRACT

In *substring compression* one is given a text to preprocess so that, upon request, a compressed substring is returned. *Generalized substring compression* is the same with the following twist. The queries contain an additional context substring (or a collection of context substrings) and the answers are the substring in compressed format, where the context substring is used to make the compression more efficient.

We focus our attention on *generalized substring compression* and present the first non-trivial *correct* algorithm for this problem. Inherent to our algorithm is a new method for finding the *bounded longest common prefix* of substrings, which may be of independent interest. In addition, we propose an efficient algorithm for substring compression which makes use of *range successor* queries.

We present several tradeoffs for both problems. For compressing the substring $S[i\,..\,j]$ (possibly with the substring $S[\alpha\,..\,\beta]$ as a context), the best query times we achieve are $O(C)$ and $O(C\log(\frac{j-i}{C}))$ for substring compression query and generalized substring compression query, respectively, where C is the number of phrases encoded.

A preliminary version of this paper has been presented in [21].

1. Introduction

While string compression has been studied for decades, substring compression is relatively lightly studied. The topic was introduced in [8], where a set of problems concerning substring compression focusing on the compression algorithm of Lempel and Ziv [35] was presented. They deal mainly with two variants of this topic, namely, given a string, what is the compressibility of different substrings of that string, both in the sense of the actual compression of the substrings and in the sense of comparing which of the substrings is the least or most compressible.

The goal of our research is to find the inherent connection between the compressed representation of a string and that of its substrings. Such a connection could have interesting practical applications, and may lead the way to finding further connections between certain properties of strings and that of their substrings. In addition, it may be interesting to further investigate the topic of substring compression using other compression methods.

We address the following problems: in the *substring compression query* (SCQ) problem, we wish to compress a given substring of the string S, denoted by start and end location. Note that we are allowed to preprocess S beforehand, so that we would be able to answer this query for any substring in S without having to scan the substring during query time. In its generalized and more powerful version, the *generalized substring compression query* (GSCQ) problem, we wish to compress

* Corresponding author.
E-mail addresses: kellero@cs.biu.ac.il (O. Keller), kopelot@cs.biu.ac.il (T. Kopelowitz), lfshir@gmail.com (S. Landau Feibish), moshe@cs.biu.ac.il (M. Lewenstein).

[1] The author participated in this research when she was a student at Bar-Ilan University.

[2] The research of the author was supported by a BSF grant 2010437, a Google Research Award and by a GIF grant 1147/2011.

0304-3975/$ – see front matter
http://dx.doi.org/10.1016/j.tcs.2013.10.010

Table 1
Results.

Problem	Query time	Space	Source
GSCQ	$O(C_{\alpha,\beta}(i,j)\log(\frac{j-i}{C_{\alpha,\beta}(i,j)}))$	$O(n^{1+\epsilon})$	new
	$O(C_{\alpha,\beta}(i,j)\log(\frac{j-i}{C_{\alpha,\beta}(i,j)})\log\log n)$	$O(n\log^{\epsilon} n)$	new
	$O(C_{\alpha,\beta}(i,j)(\log(\frac{j-i}{C_{\alpha,\beta}(i,j)})\log\log n+(\log\log n)^2))$	$O(n\log\log n)$	new
	$O\left(C_{\alpha,\beta}(i,j)\log\left(\frac{j-i}{C_{\alpha,\beta}(i,j)}\right)\log^{\epsilon} n\right)$	$O(n)$	new
SCQ	$O(C(i,j))$	$O(n^{1+\epsilon})$	new
	$O(C(i,j)\log\log n)$	$O(n\log^{\epsilon} n)$	new
	$O(C(i,j)(\log\log n)^2)$	$O(n\log\log n)$	new
	$O(C(i,j)\log^{\epsilon} n)$	$O(n)$	new
	$O(C(i,j)\log n\log\log n)$	$O(n\log^{\epsilon} n)$	[8]

the substring according to a given context taken from S as well. In both problems, our goal is to provide query times which are proportional to the size of the *compressed* substring as opposed to the size of the substring in its non-compressed form.

1.0.1. Applications

The issue of substring compression has interesting implications for a variety of practical applications. Recent works such as those presented in [7,9,31] for example, use compression of biological sequences as a basis of comparison between different sequences, and their information content. Compression of sub-sequences can therefore be used to perform such comparisons in a more efficient and accurate manner.

The result presented in [18], uses straight-line-program compression in order to speed up computation of edit distance. This result relies heavily on the findings of Rytter [32].[3] There, Rytter proved that an LZ77 [35] encoding can be transformed to a straight-line program quickly and without large expansion. Therefore, one may be able to use substring compression to speed up the edit-distance computation of substrings, which may be a problem of independent interest.

1.1. Our results

1. Our main result is an efficient and innovative algorithm for the *generalized substring compression query*, introduced in [8]. There an algorithm was suggested, however it is incorrect [29]: it overlooked the inherent added difficulty of the generalized problem which uses a bounded context, dismissing it as trivial, while it is in fact the essence of the generalized problem. The additional bounded context requires a different algorithm than the context within the substring to be compressed, that we will describe in detail in this paper. Therefore, the solution provided in [8] in fact does not solve the problem. Our solution for this problem is based on a solution to finding the *bounded longest common prefix* (BLCP) of two substrings, which is a notion we will introduce shortly.
2. In addition, we improve results shown for the *substring compression query*. Our result is based mainly on an improved solution for finding the *interval longest common prefix* (ILCP) of two substrings. This is done using an efficient solution for the problem of *range successor* [25],[4] and not on the more classical *range reporting* problem (see, for instance [1,5]), used by [8] and numerous other indexing-related papers [14,2,15,27]. This constitutes a different method in order to reduce the substring compression query problem to the geometric problem. See [26] for a survey on the connection between text indexing and various range searching techniques.

Our solutions are based on a variety of tools, such as suffix trees, lowest common ancestor queries, level-ancestor queries, and several kinds of range searching structures. As a result, solutions to both SCQ and GSCQ constitute tradeoffs between query times and space, due to the choice of range searching structures to be used. Denote C as the number of phrases encoded. A comparison of the results is presented in Table 1.

Note that range problems on strings have garnered much interest lately with different papers exploring different aspects of range problems. We refer the interested reader to [8,21,22,30,27,4,19,23].

The rest of our paper is organized as follows: in Section 2, we give some preliminaries and problem definitions. In Sections 3.3 and 3.2, we describe our solutions for finding the BLCP and ILCP. In Section 4, we present the outline of the query algorithm's main loop, which is roughly common to both the SCQ and GSCQ problems. In Sections 5 and 6, we present the solutions and analysis for SCQ and GSCQ.

[3] See also [6] where the same problem was independently addressed by Charikar et al.

[4] The range successor problem was introduced in [25] under the name *range searching for minimum*. The name "range successor" is used in [22,30]. An almost-identical problem is the *range next value* problem [12] that will be discussed later.

2. Problem definitions and preliminaries

2.1. Preliminary definitions and notations

Given a string S, $|S|$ is the length of S. Throughout this paper we denote $n = |S|$. An integer i is a *location* or a *position* in S if $i = 1, \ldots, |S|$. The substring $S[i\,..\,j]$ of S, for any two positions $i \leqslant j$, is the substring of S that begins at index i and ends at index j. Concatenation is denoted by juxtaposition. The *suffix* S_i of S is the substring $S[i\,..\,n]$.

The *suffix tree* [34,33,13,28] of a string S, denoted $\mathrm{ST}(S)$, is a compact trie of all the suffixes of $S\$$ (i.e., S concatenated with a delimiter symbol $\$ \notin \Sigma$, where Σ is the alphabet set). Each of its edges is labeled with a substring of S (actually, a representation of it, e.g., the start location and its length). The "compact" property is achieved by contracting nodes having a single child. The children of every node are sorted in the lexicographical order of the substrings on the edges leading to them. Consequently, each leaf of the suffix tree represents a suffix of S, and the leaves are sorted from left to right in the lexicographical order of the suffixes that they represent. $\mathrm{ST}(S)$ requires $O(n)$ space. Algorithms for the construction of a suffix tree enable $O(n)$ preprocessing time when $|\Sigma|$ is constant, and $O(n \log \min(n, |\Sigma|))$ time when $|\Sigma|$ is not. In fact, the suffix tree can be constructed in linear time even for alphabets drawn from a polynomially-sized range, see [13].

In addition, our algorithms make use of elements from the field of computational geometry; let $P = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ be a set of n points on an $[n] \times [n]$ grid. The following range searching query types are defined on P, for various types of a two-dimensional range R:

$\mathrm{rangesucc}_y(R = [x, x'] \times [y, \infty])$: reports the single point of P that is included in the range and has a minimal y-coordinate, i.e., the point $\arg\min_{(x,y)\in P\cap R} y$. In other words, the resulting point is the successor to y (on the y-axis) having x-coordinate in the range $[x, x']$. The query types $\mathrm{rangesucc}_x$ and $\mathrm{rangepred}_x$ for successor and predecessor respectively, this time on the x-axis, are defined in the same manner.

$\mathrm{emptiness}(R = [x, x'] \times [y, y'])$: returns "true" iff $P \cap R = \emptyset$.

2.1.1. An overview of the Lempel–Ziv algorithm

The LZ77 variation of the Lempel–Ziv algorithm works as follows: given an input string S of length n, the algorithm encodes the string in a greedy manner from left to right. At each step of the algorithm, suppose we have already encoded $S[1\,..\,k-1]$, we search for the location t, such that $1 \leqslant t \leqslant k-1$, for which the longest common prefix of $S[k\,..\,n]$ and the suffix S_t, is maximal. Once we have found the desired location, suppose the aforementioned longest common prefix is the substring $S[t\,..\,r]$, a phrase will be added to the output which will include the encoding of the distance to the substring (i.e., the value $k - t$) and the length of the substring, (i.e., the value $r - t + 1$). The algorithm continues by encoding $S[k + (r - t + 1)\,..\,n]$. Finally, we denote the output of the LZ77 algorithm on the input S as $\mathrm{LZ}(S)$.

The string S may be encoded within the context of the string T. We denote this by $\mathrm{LZ}(S \mid T)$. The practical meaning of this is that the result is as if the algorithm was performed on the concatenated string $T\$S$, where $\$$ is a symbol that does not appear in neither S nor T, however, only the portion of $\mathrm{LZ}(T\$S)$ which represents the compression of S is output by the algorithm. Some exceptions apply to this rule as will be described later. An example for the use of an additional context for compression and retrieval of genomic sequences can be seen in [24].

Recently, several works [10,20,16] have been done concerning a related problem, called *Lempel–Ziv factorization*. The Lempel–Ziv factorization of a string S is defined to be the decomposition $S = s_1 s_2 \ldots s_v$ such that for each $i = 1, \ldots, v-1$, s_i is the longest prefix of $s_i s_{i+1} \ldots s_v$ that appears in $s_1 \ldots s_{i-1}$. If this prefix is empty, s_i will be a single character [11]. It may be interesting to see how this problem and the solutions proposed for it can be adapted to the problems we present in this work.

2.2. Problem definitions

Given a string S of length n, we wish to preprocess S in such a way that allows us to efficiently answer the following queries:

Substring Compression Query ($\mathrm{SCQ}(i, j)$): given any two indices i and j, such that $1 \leqslant i \leqslant j \leqslant n$, we wish to output $\mathrm{LZ}(S[i\,..\,j])$.

Generalized Substring Compression Query ($\mathrm{GSCQ}(i, j, \alpha, \beta)$): given any four indices i, j, α, and β, such that $1 \leqslant i \leqslant j \leqslant n$ and $1 \leqslant \alpha \leqslant \beta \leqslant n$, we wish to output $\mathrm{LZ}(S[i\,..\,j] \mid S[\alpha\,..\,\beta])$.

Query times for both of the above query types will be strongly dependent on the number of phrases actually encoded. We denote these as $C(i, j)$ and $C_{\alpha,\beta}(i, j)$ for SCQ and GSCQ, respectively. Our results will rely on the two following primitives:

Bounded Longest Common Prefix ($\mathrm{BLCP}(k, l, r)$): given k, and given positions l and r which induce the context substring $S[l\,..\,r]$, we look for the longest common prefix of $S[k\,..\,j]$ and a substring which starts at some location $l \leqslant t \leqslant r$ within the context. The substring chosen must not exceed the end of context. In other words, it must be a prefix of some substring $S[t\,..\,r]$.

Interval Longest Common Prefix (ILCP(k, l, r)): given k, l, r, this time we look for the longest common prefix of $S[k\,..\,j]$ and a substring which starts at some location $l \leqslant t \leqslant r$, without further constraints.k

While it may not seem so at first glance, BLCP queries are more difficult to implement than ILCP. For example, consider two suffixes S_{t_1} and S_{t_2}, such that $l \leqslant t_1 < t_2 \leqslant r$, for which $|\text{LCP}(S_k, S_{t_1})| < |\text{LCP}(S_k, S_{t_2})|$ (where LCP(S_1, S_2), for two strings S_1 and S_2, stands for the *longest common prefix* of S_1 and S_2). Some portion of the last characters of LCP(S_k, S_{t_2}) may not be eligible for consideration. Namely, if $|\text{LCP}(S_k, S_{t_2})|$ exceeds $r - t_2 + 1$ characters, LCP(S_k, S_{t_2}) exceeds location r, and therefore literally "grows out of context". In that case, it may be that S_{t_1} will eventually be the suffix to be preferred. One should take into account that such a cut-off may pertain to LCP(S_k, S_{t_1}) as well. (Note: in the case $i - 1 \leqslant r < j$, if desired, one can allow a substring taken from the context to exceed r. This is a trivial extension to the algorithm for ILCP.)

3. Limiting the longest common prefix: answering ILCP and BLCP queries

3.1. Preprocessing motivation

We begin the preprocessing by constructing the suffix tree of S, ST(S). In the suffix tree, each leaf ℓ is associated with a suffix of $S\$$ and is therefore marked with an integer $y(\ell)$ which is the start location of that suffix. We also mark each leaf ℓ with an integer $x(\ell)$ which is the lexicographical rank of the suffix associated with ℓ within the set of all suffixes of T (this is done by using one depth-first traversal, in which we number the leaves from left to right). We then preprocess the set $P = \{(x(\ell), y(\ell)) \mid \ell \text{ is a leaf in ST}(S)\} \subseteq [n+1]^2$ for the range searching query types mentioned before. We will refer to the points in P as *suffix points*.

Suppose we search ST(S) for some substring $S[l\,..\,r]$, we can find all the occurrence positions of $S[l\,..\,r]$ in S, by traversing ST(S) from the root downwards according to the symbols in $S[l\,..\,r]$, until either (1) the next symbol of the pattern cannot be found at our current location in the tree—in this case we conclude that $S[l\,..\,r]$ does not occur in S; (2) the pattern is exhausted and we conclude the traversal at a node v in ST(S) (or the edge leading to it from its parent, for that matter). Let v be the node in which the search ended. All the leaves in the subtree rooted at v, denoted T_v, correspond to occurrences of $S[l\,..\,r]$ in S. Hence the set $Y_v = \{y(\ell) \mid \ell \text{ is a leaf in } T_v\}$ is the set of all occurrence positions of $S[l\,..\,r]$ in S. From the properties of the suffix tree it follows that the set $X_v = \{x(\ell) \mid \ell \text{ is a leaf in } T_v\}$ forms a consecutive range of values in $[n+1]$. This is exactly the range $X_v = [x(l_v), x(r_v)]$, where l_v and r_v are the leftmost and rightmost leaves in T_v, respectively. It therefore holds that for a leaf ℓ, ℓ is a leaf in T_v iff $x(\ell) \in [x(l_v), x(r_v)]$. In other words: $x(\ell) \in [x(l_v), x(r_v)]$ iff $S[l\,..\,r]$ appears in S at location $y(\ell)$.

Notice that each node u in the suffix tree has two different notions of *depth*: the ordinary perception of depth of a node in a tree, denoted depth(u), and the length of the string u represents (derived by the concatenation of edge labels on the path from the root to u), denoted length(u). Now let S_i and S_j be two suffixes of S, and consider the *longest common prefix* of S_i and S_j, denoted LCP(S_i, S_j). Let ℓ_i and ℓ_j be the leaves corresponding to S_i and S_j, respectively (i.e., $i = y(\ell_i)$ and $j = y(\ell_j)$). Then $|\text{LCP}(S_i, S_j)| = \text{length}(\text{LCA}(\ell_i, \ell_j))$, where LCA$(\ell_i, \ell_j)$ is the *lowest common ancestor* of ℓ_i and ℓ_j.

3.2. Answering ILCP queries

Here our primary goal is to obtain an efficient way of finding ILCP(k, l, r), that is, given k, l, r, we seek the longest common prefix of $S[k\,..\,j]$ and a substring which starts at some location $l \leqslant t \leqslant r$. An example of this constraint can be seen in Fig. 1.

Recall that when searching for ILCP(k, l, r), while the resulting substring must start at some location $l \leqslant t \leqslant r$, it is allowed to exceed location r. This is the equivalent to finding the location $l \leqslant t \leqslant r$, for which the longest common prefix of $S[k\,..\,j]$ and the suffix S_t, is maximal.

Consider the suffix S_k. Clearly, it is sufficient to find the location $t \in [l, r]$ for which $|\text{LCP}(S_k, S_t)|$ is maximized, i.e., $t = \arg\max_{z \in [l,r]} |\text{LCP}(S_k, S_z)|$ (without necessarily computing the value $|\text{LCP}(S_k, S_t)|$ at this stage). Once the aforementioned location t is found, we compute $|\text{LCP}(S_k, S_t)|$. Therefore, to summarize, we have two steps: (1) finding the location t, and (2) computing $|\text{LCP}(S_k, S_t)|$.

3.2.1. Finding the start location t

We use a reduction to the problem of range successor on a grid, as opposed to the *range reporting* used in [8]. An example of the geometric representation of the scenario depicted in Fig. 1 can be seen in Fig. 2.

Consider the suffix S_k, and consider the set of suffixes $\Gamma = \{S_l, \ldots, S_r\}$. Since $|\text{LCP}(S_k, S_t)| = \max_{z \in [l,r]} |\text{LCP}(S_k, S_z)|$, S_t is in fact the suffix lexicographically closest to S_k, out of all the suffixes of the set Γ.

We will first assume that we are searching for a suffix S_{t_1}, such that the suffix S_{t_1} is *lexicographically smaller* than S_k. The process for the case where the suffix chosen is lexicographically greater than S_k is symmetric. Therefore, all we are required is to choose the best of both, i.e., the option yielding the greater $|\text{LCP}(S_k, S_t)|$ value.

Since we have assumed w.l.o.g. that S_{t_1} is lexicographically smaller than S_k, we have actually assumed that $x(\ell_{t_1}) < x(\ell_k)$, or equivalently, that ℓ_{t_1} appears to the *left* of ℓ_k in the suffix tree. Incorporating the lexicographical ranks of S_k and S_{t_1} into

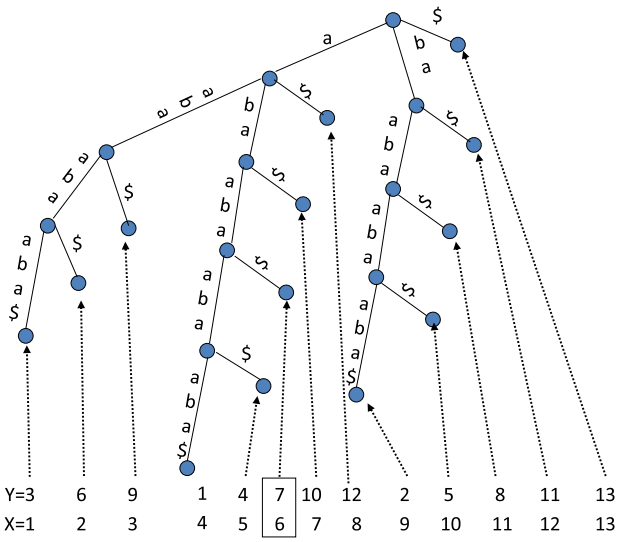


Fig. 1. The suffix tree for the string $S = abaabaabaaba\$$ is shown. The x values indicate the lexicographical rank of the suffix. The y values indicate the index of the suffix in S. Assume we are compressing the substring: $S[4\,..\,9]$ that is *abaaba*, and that, so far, we have encoded the substring: $S[4\,..\,6]$, meaning **aba***aba*. The marked suffix indicates the suffix that should be encoded next.

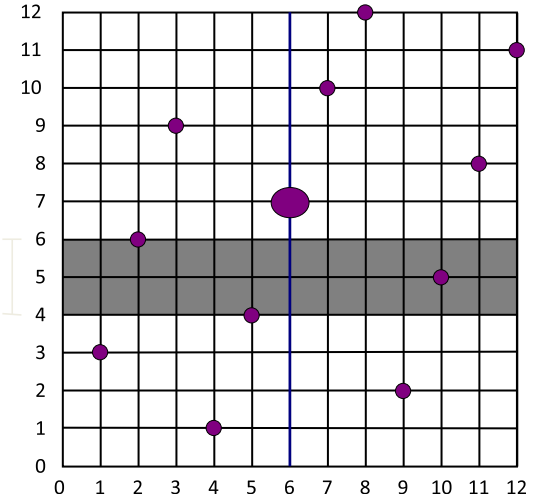


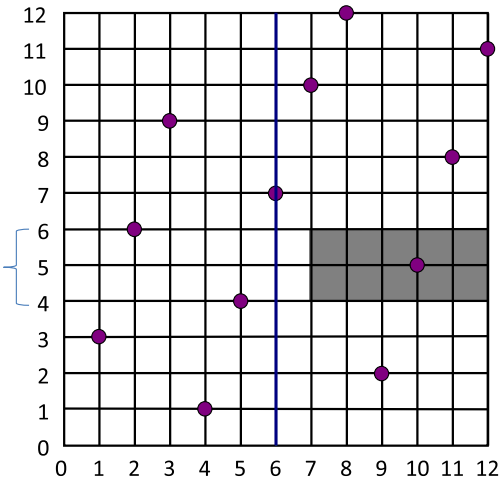
Fig. 2. The grid depicts the geometric representation of the suffix tree for the string $S = abaabaabaaba\$$. The enlarged point at $(6, 7)$ represents the suffix S_7, as the substring *aba*, yet to be encoded, starts at position 7 in the string S. The grayed area of the grid represents the part of the substring which has already been encoded, that is $S[4\,..\,6]$. When finding the start location t, we will be limited to using points found in the gray area.

the expression, t_1 is actually the value which maximizes the expression $\max\{x(\ell_{t_1}) \mid l \leqslant t_1 \leqslant r \text{ and } x(\ell_{t_1}) < x(\ell_k)\}$. Notice that $t_1 = y(\ell_{t_1})$.

Now consider the set $P = \{(x(\ell), y(\ell)) \mid \ell \text{ is a leaf in } \mathrm{ST}(S)\}$. Assuming indeed $x(\ell_{t_1}) < x(\ell_k)$, we are interested in finding the maximal value $x(\ell_{t_1})$, such that $x(\ell_{t_1}) < x(\ell_k)$, and $l \leqslant y(\ell_{t_1}) \leqslant r$. It immediately follows that the point $(x(\ell_{t_1}), y(\ell_{t_1})) \in P$ is the suffix point in the range $[-\infty, x(\ell_k) - 1] \times [l, r]$ having maximal x-coordinate, and therefore can be obtained efficiently by querying $\mathrm{rangepred}_x([-\infty, x(\ell_x) - 1] \times [l, r])$. Once we have found the point $(x(\ell_{t_1}), y(\ell_{t_1}))$, we can locate ℓ_{t_1}, as it is the $x(\ell_{t_1})$-th leaf from the left. The leaf ℓ_{t_1} will be of importance in the next section.

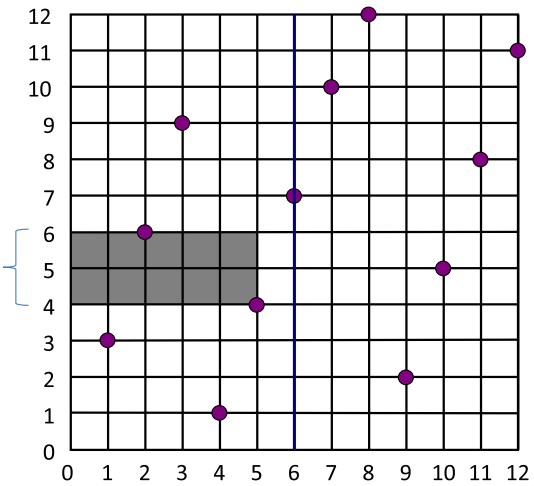
Equivalently, there exists t_2 such that S_{t_2} is the suffix lexicographically larger than S_k and closest to it. In other words, we assume $x(\ell_{t_2}) > x(\ell_k)$, or equivalently, that ℓ_{t_2} appears to the *right* of ℓ_k in the suffix tree. t_2 can be found using a symmetric procedure: we query for the $\mathrm{rangesucc}_x([x(\ell_x) + 1, \infty] \times [l, r])$. An example of the queries performed can be seen in Fig. 3.

Determining whether $t = t_1$ or $t = t_2$ is implemented by calculating both $|\mathrm{LCP}S_k, S_{t_1}|$ and $|\mathrm{LCP}S_k, S_{t_2}|$, and choosing the larger of the two. The exact method of calculating both values is described next.



(a) $\text{rangesucc}_x([x(\ell_x)+1,\infty]\times[l,r])$,

Output: the point $(10,5)$.



(b) $\text{rangepred}_x([-\infty,x(\ell_x)-1]\times[l,r])$,

Output: the point $(5,4)$.

Fig. 3. Each grid depicts the geometric representation of the suffix tree for the string $S = abaabaabaaba\$$. The values given for the example queries are: $x = 6$, $l = 4$ and $r = 6$. The chosen suffix in this case would be S_4 since $|\text{LCP}(S_7, S_4)|$ is greater than $|\text{LCP}(S_7, S_5)|$.

3.2.2. Computing $|\text{LCP}(S_k, S_t)|$

Consider ℓ_k and ℓ_t as described above. Since $|\text{LCP}(S_i, S_j)| = \text{length}(\text{LCA}(\ell_i, \ell_j))$ for any i and j, it is sufficient to find the node $w = \text{LCA}(\ell_k, \ell_t)$ and then to compute length(w). Using the methods of Harel and Tarjan [17], an LCA query can be answered in constant time. If the value length(u) for each node u has been stored in u beforehand, we conclude the value length(w) is obtainable in $O(1)$ time.

3.3. Answering BLCP queries

Assume that we wish to answer the query $\text{BLCP}(k, l, r)$. Consider the suffix S_k represented by the path from the root to ℓ_k. For any other suffix S_t and an integer $d \geqslant 0$, let $\text{LCP}_d(S_k, S_t)$ be the longest common prefix of S_k and S_t, truncated to at most d characters (i.e., if $\text{LCP}(S_k, S_t)$ exceeds d characters, we will leave only the first d and discard the others). By the definition of $\text{BLCP}(k, l, r)$, we are limited to finding substrings that do not exceed location r. Therefore, we actually wish to find $t' = \arg\max_{t\in[l,r]} |\text{LCP}_{r-t+1}(S_k, S_t)|$. Here notice that the term $r - t + 1$ is the maximal length of the portion of S_t we can use, according to the constraints. Also notice that by definition, $|\text{LCP}_{r-t+1}(S_k, S_t)| = \min\{|\text{LCP}(S_k, S_t)|, r - t + 1\}$. If several positions t that maximize the above expression exist, we define t' to be the leftmost such position, i.e., the smallest such value.

Definitions and notations. For a node u, let path(u) be the path in ST(T) from the root to u. With a slight abuse of notation, we will also use path(u) to denote the set of nodes participating in such a path. For a suffix S_t we also define $\text{path}(S_t) = \text{path}(\ell_t)$ where ℓ_t is the leaf representing S_t in the suffix tree. In the context of a $\text{BLCP}(k, l, r)$ query, a suffix S_t is said to be *relevant* if $l \leqslant t \leqslant r$.

A suffix S_t is said to be *eligible* at a node $u \in \text{path}(S_k)$ if the string represented by path(u) is a prefix of $\text{LCP}_{r-t+1}(S_k, S_t)$. Consider the suffix S_k represented by the path from the root to ℓ_k. As suffixes S_t with greater $|\text{LCP}(S_k, S_t)|$ values branch out of this path at a later stage (i.e., leave this path at nodes of greater depth), we are interested in suffixes which share a large portion of this path. However, as $|\text{LCP}_{r-t+1}(S_k, S_t)|$ (and not $|\text{LCP}(S_k, S_t)|$) is the expression to be maximized, we are restricted by the eligibility of suffixes along nodes in path(S_k). An example of eligibility is shown in Fig. 4.

We shall find t' in two phases. First we will efficiently locate a node $v \in \text{path}(S_k)$ of maximal depth such that there exists a relevant suffix S_t which is eligible at v. We will in fact prove that $S_{t'}$ is one of the suffixes that are eligible at v. Second, in the case that there are several relevant suffixes S_t which are eligible at v, we shall efficiently find $S_{t'}$ among them.

3.3.1. Finding node v

Observe path(S_k). We wish to search this path for the lowest node v for which there exists t such that S_t is relevant and eligible at v. Notice that the notion of eligibility satisfies the property that if some suffix S_t is eligible at some node $u \in \text{path}(S_k)$, then S_t is eligible at all of u's ancestors as well. In other words, the eligibility property is monotone along path(S_k). If the suffix tree had been preprocessed for answering level-ancestor queries, by the methods of, for example, [3],

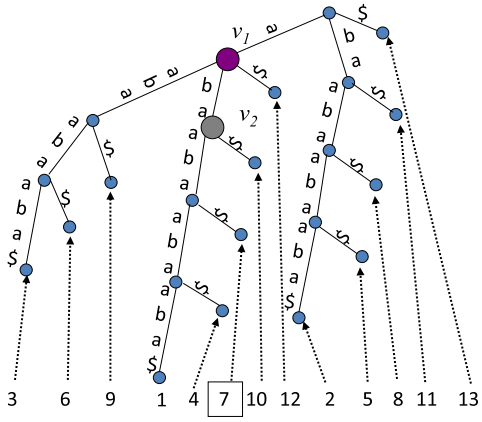


Fig. 4. Again we look at the suffix tree of the string $S = abaabaabaaba\$$. This time we would like to encode the substring $S[7..9]$ within the context of $S[2..5]$. For this example, looking at location $r = 5$, the suffix S_4 is eligible at node v_1 and is not eligible at node v_2.

we can find the ancestor of ℓ_k of a specific depth d in $O(1)$ time. We conclude that we can perform a binary search on the depth of nodes on this path: at each node u we probe, we will efficiently test whether there exists some relevant suffix which is eligible at u.

We can perform this test by conducting the following range emptiness query: emptiness$([x(l_u), x(r_u)] \times [l, r - \text{length}(u) + 1])$. This is captured by the two following lemmas:

Lemma 1. *A suffix S_t is eligible at a node $u \in \text{path}(S_k)$ iff $u \in \text{path}(S_t)$ and $\text{length}(u) \leqslant r - t + 1$.*

Proof. Let S^u be the string represented by path(u). By definition, S_t is eligible at u if S^u is a prefix of $\text{LCP}_{r-t+1}(S_k, S_t)$. That happens if and only if (a) S^u is a prefix of $\text{LCP}(S_k, S_t)$ and therefore $u \in \text{path}(S_t)$, and (b) the length of S^u, i.e., length(u) is at most $r - t + 1$. □

Lemma 2. *Fix the range $[l, r]$ and let S_t be a suffix. Then $(x(\ell_t), y(\ell_t)) \in ([x(l_u), x(r_u)] \times [l, r - \text{length}(u) + 1])$ iff S_t is relevant and eligible at u.*

Proof. Let S_t be a suffix. From the properties of a suffix tree, it holds that S_t's lexicographical rank $x(\ell_t)$ is in $[x(l_u), x(r_u)]$ if and only if ℓ_t is a leaf in T_u or equivalently if and only if $u \in \text{path}(S_t)$. For its start location $t = y(\ell_t)$, $y(\ell_t) \in [l, r - \text{length}(u) + 1]$ if and only if both $y(\ell_t) \in [l, r]$ (i.e., S_t is relevant) and $y(\ell_t) \leqslant r - \text{length}(u) + 1$. As the latter can be re-stated as $\text{length}(u) \leqslant r - t + 1$, the lemma holds. □

We conclude that the emptiness query returns a negative result if and only if there exists a suffix S_t which is both relevant and eligible at u.

Instead of the ordinary $O(\log n)$-time binary search, we use a mixed "galloping" and ordinary binary search approach: we conduct the search by iterations, where in the i-th iteration we probe the node on the path whose depth is $2^{i-1} - 1$ and conduct the proper range emptiness query on it, repeating this process until the first node whose emptiness query returned a positive result is encountered. Denote this node as q and denote the last node probed before q as p. Now we find v by binary searching on the sub-path between p and q. The main importance of the mixed search, is that now, using a refined analysis, we will later prove that encoding a phrase is done in time logarithmic in the phrase's length (rather than n).

Recall that we have defined $t' = \arg\max_{t \in [l,r]} |\text{LCP}_{r-t+1}(S_k, S_t)|$. At the end of this phase we have the following lemma:

Lemma 3. *v is the node of maximal depth at which $S_{t'}$ is eligible.*

Proof. Assume not, and let $v' \neq v$ be the node of maximal depth at which $S_{t'}$ is eligible. (Notice that v' is well-defined since $S_{t'}$ is trivially eligible at the root.) Since both $v, v' \in \text{path}(S_k)$ then either v is an ancestor of v' or vice-versa.

In the former case, $\text{depth}(v') > \text{depth}(v)$. However, by Lemma 2 and the monotonicity of the eligibility property, the binary search described before must conclude with the maximal depth node at which some relevant suffix is eligible, which contradicts the fact that $S_{t'}$ is eligible at v'.

In the latter case $\text{length}(v') < \text{length}(v)$ and by Lemma 2, there exists a relevant suffix S_t which is eligible at v, and therefore $|\text{LCP}_{r-t+1}(S_k, S_t)| \geqslant \text{length}(v)$. Since $\text{length}(v') < \text{length}(v)$ and by the definition of v', $S_{t'}$ cannot be eligible at v. Therefore, $|\text{LCP}_{r-t'+1}(S_k, S_{t'})| < \text{length}(v)$, otherwise, when traversing $\text{LCP}_{r-t'+1}(S_k, S_{t'})$ starting from the root of the suffix

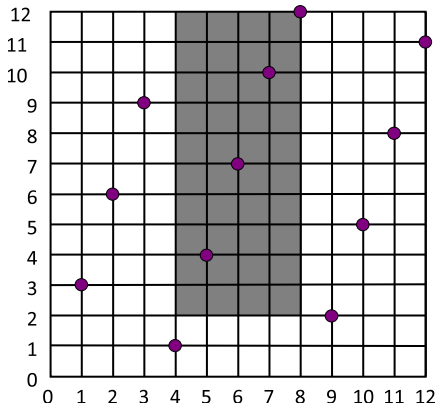


Fig. 5. In Fig. 4 we found $v = v_1$, since v_1 is the node of maximal depth on the path from the root to S_7, at which S_7 is eligible w.r.t. $r = 5$. Therefore, the values of our range will be: $x(l_v) = 4$, $x(r_v) = 8$ and $l = 2$. Therefore, we obtain the range query as depicted by the grayed area. The output point of this query will be $(5, 4)$.

tree, we would have to visit v, a fact which would make $S_{t'}$ eligible at v as well. Therefore $|\text{LCP}_{r-t+1}(S_k, S_t)| \geqslant \text{length}(v) > |\text{LCP}_{r-t'+1}(S_k, S_{t'})|$, which contradicts the optimality of t'. □

3.3.2. Finding the suffix $S_{t'}$

Recall that for a suffix S_t, $|\text{LCP}_{r-t+1}(S_k, S_t)| = \min\{|\text{LCP}(S_k, S_t)|, r - t + 1\}$, and therefore this value can be computed in $O(1)$ using a single LCA query. Given v, as described in the previous phase, let $w \in \text{path}(S_k)$ be its specific child that is on $\text{path}(S_k)$. We then perform the following: we inspect v and query $\text{rangesucc}_y([x(l_v), x(r_v)] \times [l, \infty])$. Let (x', y') be the point returned by the query (see example in Fig. 5). We also inspect w and perform $\text{rangesucc}_y([x(l_w), x(r_w)] \times [l, \infty])$, and let (x'', y'') be the resulting point, if such exists. If (x'', y'') does not exist or that $y'' \notin [l, r - \text{length}(v) + 1]$, we choose t' to be y'. Otherwise, notice that y', y'' are the two start positions of the relevant suffixes $S_{y'}, S_{y''}$ respectively. Pick the value $t \in \{y', y''\}$ that maximizes the expression $|\text{LCP}_{r-t+1}(S_k, S_t)|$ and choose it to be t' (if both maximize the expression, choose $t' = \min\{y', y''\}$). Finally return the resulting substring, as represented by its start position t' and its length $|\text{LCP}_{r-t'+1}(S_k, S_{t'})|$. A pseudo-code of the above process can be seen in Procedure Compute-t'.

Procedure Compute-t'.

```
   Input: v
   Output: selected t'
 1 w ← v's child on path(S_k);
 2 (x', y') ← rangesucc_y([x(l_v), x(r_v)] × [l, ∞]);
 3 (x'', y'') ← rangesucc_y([x(l_w), x(r_w)] × [l, ∞]);
 4 if (x'', y'') does not exist or y'' ∉ [l, r − length(v) + 1] then
 5 |  t' ← y';
 6 else
 7 |  if |LCP_{r−y'+1}(S_k, S_{y'})| = |LCP_{r−y''+1}(S_k, S_{y''})| then
 8 |  |  t' ← min{y', y''};
 9 |  else
10 |  |  t' ← arg max_{t∈{y',y''}} |LCP_{r−t+1}(S_k, S_t)|;
11 return t';
```

Before formally proving the correctness of the above procedure, we provide some intuition for it. Observe node v once again and the range $[x(l_v), x(r_v)] \times [l, r - \text{length}(v) + 1]$. There might be several relevant suffixes which are eligible at v, and therefore have their corresponding suffix points in that range. In this case, for each such suffix S_t, v is the node of maximal depth at which S_t is eligible. Notice that some of those suffixes may be represented by paths that branch out from v to w, and some by paths that branch out from v to one of its other children. Therefore, their potential "contribution to compression" may be different: for the specific case where for a suffix S_t, the path from the root to ℓ_t visits v and then w, and also it holds that $t + \text{length}(v) - 1 < r$, there is an additional eligible portion of S_t of length $r - (t + \text{length}(v) - 1)$ on the edge (v, w) (figuratively speaking; we mean of course that the additional eligible portion is a prefix of the substring represented by the label of (v, w)). We refer to such suffixes as *special suffixes* and note that the additional portion may be of a different length for different special suffixes. An example of the special suffix scenario is shown in Fig. 6. The existence of such special suffixes creates the need to inspect both v and w.

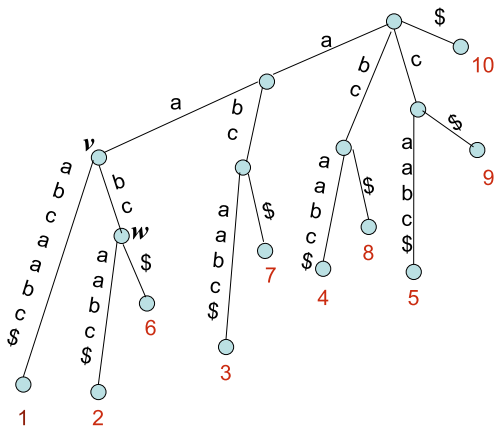


Fig. 6. In this example, the suffix tree for the string $S = aaabcaabc\$$ is shown. Assume we wish to compress $S[6\,..\,9]$ (i.e., $k = 6$) and the given context is $S[1\,..\,4]$ (i.e., $r = 4$). Nodes v and w are labeled accordingly. Notice that the leftmost relevant suffix in T_v is $S_{y'} = S_1$ for which $|\mathrm{LCP}_{r-y'+1}(S_k, S_{y'})| = \min\{\mathrm{length}(v), r - y' + 1\} = 2$. However, the leftmost relevant suffix in T_w is $S_{y''} = S_2$ for which $|\mathrm{LCP}_{r-y''+1}(S_k, S_{y''})| = r - y'' + 1 = 3$. Therefore we would like to output start location 2.

We move to the formal proof:

Theorem 1. *The value returned by the above procedure is* $t' = \arg\max_{t\in[l,r]} |\mathrm{LCP}_{r-t+1}(S_k, S_t)|$.

Proof. Consider all relevant suffixes that are eligible at v, and therefore are represented by suffix points in the range $[x(l_v), x(r_v)] \times [l, r - \mathrm{length}(v) + 1]$ (by Lemma 2). By Lemma 3, $S_{t'}$ is among these suffixes, however we do not know whether $\mathrm{path}(S_{t'})$ visits w, or not.

Notice that for all such suffixes S_t which branch out from v to a node other than w, it holds that $|\mathrm{LCP}_{r-t+1}(S_k, S_t)| = \min\{|\mathrm{LCP}(S_k, S_t)|, r - t + 1\} = |\mathrm{LCP}(S_k, S_t)| = \mathrm{length}(v)$. As these suffixes all have the same $|\mathrm{LCP}_{r-t+1}(S_k, S_t)|$ value, our first $\mathrm{rangesucc}_y$ query simply picks the one with the leftmost start position.

On the other hand, for all such suffixes S_t which branch out from v to w, it holds that $|\mathrm{LCP}(S_k, S_t)| > r - t + 1$ (since for them, $|\mathrm{LCP}(S_k, S_t)| \geqslant \mathrm{length}(w)$, but they are not eligible at w) and therefore $|\mathrm{LCP}_{r-t+1}(S_k, S_t)| = r - t + 1$. In other words, among these suffixes, the one maximizing $|\mathrm{LCP}_{r-t+1}(S_k, S_t)|$ is the one with the leftmost start position. As all these suffixes have a one-to-one correspondence with suffix points in $[x(l_w), x(r_w)] \times [l, r - \mathrm{length}(v) + 1]$ (by Lemma 2 and the fact that those suffixes have paths that visit w), our goal is to pick the suffix point in the range having the minimal y-coordinate. This is done by our second $\mathrm{rangesucc}_y$ query with the additional test that for the resulting point (x'', y'') (if such exists), indeed $y'' \in [l, r - \mathrm{length}(v) + 1]$.

Since at first we do not know which of the cases holds for t', the algorithm simply chooses the best of both results. □

4. Outline of substring compression query algorithms

Given locations i and j which induce the substring $S[i\,..\,j]$ to be compressed, we describe the outline of our methods, in an inductive manner:

- For the first location i, two cases exist, according to query type:
  SCQ: write the encoded representation of $S[i]$.
  GSCQ: set $k \leftarrow i$ and calculate $\mathrm{BLCP}(k, \alpha, \beta)$. For convenience, we denote $|\mathrm{LCP}| = |\mathrm{BLCP}(k, \alpha, \beta)|$.
- For a general location, assume $S[k\,..\,j]$ is left to be compressed. Again two cases exist:
  SCQ: the LZ method revolves around finding $\mathrm{ILCP}(k, i, k - 1)$. For convenience, we denote $\mathrm{LCP} = \mathrm{ILCP}(k, i, k - 1)$.
  GSCQ: here we calculate both $\mathrm{ILCP}(k, i, k - 1)$ and $\mathrm{BLCP}(k, \alpha, \beta)$, and choose the longest of both. For convenience, this time we denote $|\mathrm{LCP}| = \max\{|\mathrm{ILCP}(k, i, k - 1)|, |\mathrm{BLCP}(k, \alpha, \beta)|\}$.

It is important to note that in all cases we need not find the LCP itself, but rather it is sufficient to find its starting position t and its length. Once the proper $|\mathrm{LCP}|$ value is obtained, if $k + |\mathrm{LCP}| - 1 > j$, we truncate its last characters, leaving only the first $j - k + 1$.

If no such LCP exists (e.g., $|\mathrm{ILCP}| = 0$ and, if applicable, $|\mathrm{BLCP}| = 0$), we revert to writing the encoded representation of the current character, i.e., $S[k]$. Otherwise, we write the encoded representation of the distance to the starting position t (i.e., the value $k - t$) and length of LCP, and set $k \leftarrow k + |\mathrm{LCP}|$. If $k \leqslant j$, we repeat this process, otherwise, we stop.

Table 2
SCQ tradeoffs.

rsucc/rpred	Query time	Space
[30]	$O(C(i,j)\log\log n)$	$O(n\log^{\epsilon} n)$
[30]	$O(C(i,j)(\log\log n)^2)$	$O(n\log\log n)$
[30]	$O(C(i,j)\log^{\epsilon} n)$	$O(n)$
[12]	$O(C(i,j))$	$O(n^{1+\epsilon})$

5. Substring compression query

Given a string $S[1..n]$, it will be preprocessed to efficiently answer queries of the form SCQ(i,j), in which we are asked to find the compression of the substring $S[i..j]$. The compression of $S[i..j]$ will then be computed by performing ILCP queries in the manner described above until the compressed representation of the entire substring has been found.

5.1. Analysis

Our running times and space used are heavily affected by the choice of the range searching structure used. The following presents the general tradeoff scheme:

Theorem 2. SCQ(i,j) *can be answered in worst-case* $O(C(i,j)Q_{\text{rsucc}})$ *time, using a structure which employs* $O(S_{\text{rsucc}})$ *space, where* Q_{rsucc} *and* S_{rsucc} *stand for the query time and space of the range successor structure, respectively.*

Proof. We analyze the space and query time:

Space. Consists of: $O(n)$ for the suffix tree, augmented with the additional $x(\ell)$ and length(u) values, and LCA information; S_{rsucc} for the range successor structure. Since in all of our configurations, the space for the range successor structure dominates the space requirements, we conclude the space used is $O(S_{\text{rsucc}})$.

Query time. For each of the $C(i,j)$ phrases encoded, we use: Q_{rsucc} for range predecessor (resp. successor) queries made in order to find ℓ_{t_1} (resp. ℓ_{t_2}); $O(1)$ in order to compute both $|\text{LCA}(\ell_k,\ell_{t_1})|$ and $|\text{LCA}(\ell_k,\ell_{t_2})|$, and choose the maximum of both. We conclude the query time is overall $O(C(i,j)Q_{\text{rsucc}})$. □

Substituting in the different flavors for range successor data structure of Nekrich and Navarro [30], we get the query time–space tradeoffs presented in Table 2.

We also provide the following tradeoff:

Theorem 3. *For any constant* $\epsilon > 0$, SCQ(i,j) *can be answered in worst-case* $O(C(i,j))$ *time, using a structure which employs* $O(n^{1+\epsilon})$ *space.*

Proof. Notice that our range queries are performed on $x(\ell)$ and $y(\ell)$ values. A unique property of these values is that no $x(\ell)$ or $y(\ell)$ value occurs twice in P, i.e., the sequence of point x-coordinates, and the sequence of point y-coordinates, are both permutations of $[n+1]$. Using the *range next value* structure of [12] allows us to obtain the following tradeoff: the space used is dominated by the $O(n^{1+\epsilon})$ space required for the range successor structure, and for the query time, since a single range successor/predecessor query can now be answered in $O(1)$, the overall query time is worst-case $O(C(i,j))$. □

6. General substring compression query

For GSCQ, in addition to the two locations i and j, which denote the substring $S[i..j]$ to be compressed, we receive two more indices α and β, which induce a *context* substring $S[\alpha..\beta]$. This time we are asked to provide LZ$(S[i..j] \mid S[\alpha..\beta])$.

Here, when trying to compress $S[k..j]$ for some $i \leqslant k \leqslant j$, we have two options: for the first we consider phrases having a start position $i \leqslant t \leqslant k-1$. This option is the one solved in Section 5, using ILCP queries. The second, is to consider phrases taken from $S[\alpha..\beta]$. This will be done using a BLCP query.

6.1. Analysis

The analysis is depicted in the following theorem:

Theorem 4. GSCQ(i,j,α,β) *can be answered in worst-case*

$$O\left(C_{\alpha,\beta}(i,j)\left(\log\left(\frac{j-i}{C_{\alpha,\beta}(i,j)}\right)Q_{\text{empt}} + Q_{\text{rsucc}}\right)\right)$$

Table 3
GSCQ tradeoffs.

empt	rsucc	Query time	Space
[12]	[12]	$O(C_{\alpha,\beta}(i,j)\log(\frac{j-i}{C_{\alpha,\beta}(i,j)}))$	$O(n^{1+\epsilon})$
[5]	[30]	$O(C_{\alpha,\beta}(i,j)\log(\frac{j-i}{C_{\alpha,\beta}(i,j)})\log\log n)$	$O(n\log^{\epsilon} n)$
[5]	[30]	$O(C_{\alpha,\beta}(i,j)(\log(\frac{j-i}{C_{\alpha,\beta}(i,j)})\log\log n+(\log\log n)^2))$	$O(n\log\log n)$
[30]	[30]	$O(C_{\alpha,\beta}(i,j)\log(\frac{j-i}{C_{\alpha,\beta}(i,j)})\log^{\epsilon} n)$	$O(n)$

time, using a structure which takes $O(S_{\text{empt}}+S_{\text{rsucc}})$ space, where Q_{empt} and S_{empt} (resp. Q_{rsucc} and S_{rsucc}) are the query time and space of the range emptiness (resp. range successor) structure, respectively.

Proof. As follows:

Space. Consists of: $O(n)$ for the suffix tree, augmented with the additional $x(l_u)$, $x(r_u)$ and length(u) values, LCA and level-ancestor structure information. These bounds will be dominated by the range searching structures chosen.

Query time. Consider the query's main loop described in Section 4 and consider the d-th iteration of the query algorithm main loop, and let len_d be the length of the phrase encoded in this iteration ($d=1,\ldots,C_{\alpha,\beta}(i,j)$). Assume $S[k\,..\,j]$ is the portion left to be compressed just before this iteration, and let nodes v, p, and q be as defined before. It holds that $\text{depth}(p)\leqslant\text{length}(p)\leqslant\text{length}(v)\leqslant|\text{BLCP}(k,\alpha,\beta)|$. Node q was found one iteration after node p. Therefore:

$$\text{depth}(q)\leqslant 2\big(\text{depth}(p)+1\big)\leqslant 2\big(\big|\text{BLCP}(k,\alpha,\beta)\big|+1\big). \tag{1}$$

We conclude that finding q was done by performing $O(\log|\text{BLCP}(k,\alpha,\beta)|)$ node accesses, and the following binary search, was supported by performing

$$O\big(\log\big(\text{depth}(q)-\text{depth}(p)\big)\big)=O\big(\log\big|\text{BLCP}(k,\alpha,\beta)\big|\big) \tag{2}$$

node accesses. Since

$$\big|\text{BLCP}(k,\alpha,\beta)\big|\leqslant\max\big\{\big|\text{ILCP}(k,i,k-1)\big|,\big|\text{BLCP}(k,\alpha,\beta)\big|\big\}=\text{len}_d, \tag{3}$$

and when accessing each node, a range emptiness query was conducted, overall time for the mixed search described is $O(\log(\text{len}_d)\cdot Q_{\text{empt}})$, where Q_{empt} is the query time used for the emptiness query. We conclude that a BLCP(k,α,β) query can be answered in $O(\log(\text{len}_d)\cdot Q_{\text{empt}}+Q_{\text{rsucc}})$, where Q_{rsucc} is the time required for each of the two final range successor queries performed. Recall that an ILCP$(k,i,k-1)$ query is also made, however, this query only takes $O(Q_{\text{empt}})$ time, and therefore does not influence the time bound.
We conclude that GSCQ can be answered in overall

$$O\left(Q_{\text{empt}}\sum_{d=1}^{C_{\alpha,\beta}(i,j)}\log(\text{len}_d)+C_{\alpha,\beta}(i,j)Q_{\text{rsucc}}\right) \tag{4}$$

time. $\{\text{len}_d\}_{d=1}^{C_{\alpha,\beta}(i,j)}$ is a partition of $|S[i\,..\,j]|=j-i+1$, therefore the above expression is maximized when $\text{len}_1=\cdots=\text{len}_{C_{\alpha,\beta}(i,j)}=\frac{j-i+1}{C_{\alpha,\beta}(i,j)}$. We conclude that GSCQ$(i,j,\alpha,\beta)$ can be answered in

$$O\left(C_{\alpha,\beta}(i,j)\left(\log\left(\frac{j-i}{C_{\alpha,\beta}(i,j)}\right)Q_{\text{empt}}+Q_{\text{rsucc}}\right)\right) \tag{5}$$

time. □

The choice of range emptiness and range successor structures will determine the time bounds for their respective queries. Tradeoff results are given in Table 3, where the column labeled "empt" denotes the range emptiness structure used, and the column labeled "rsucc" denotes the range successor structure used.

7. Conclusion

The goal of our research was to find the inherent connection between the compressed representation of a string and that of its substrings. We have based our research on the Lempel–Ziv algorithm and focused our attention on two main variants of the problem: basic substring compression, and generalized substring compression. For these problems we have

achieved the following query times: given a string S, for a chosen substring $S[i\,..\,j]$ (possibly with the substring $S[\alpha\,..\,\beta]$ as a context), assuming C phrases are needed for the compressed representation of $S[i\,..\,j]$ using the LZ77 algorithm, the best query times we achieve are $O(C)$ and $O(C\log(\frac{j-i}{C}))$ for the basic substring compression query and the generalized substring compression query, respectively. While initial results for these problems were presented in [8], our results are an improvement for the basic substring compression query, and are the first known correct ones for the generalized substring compression query. The problems we have dealt with, and the solutions proposed, leave us with two main problems that may each be of independent interest.

1. The main problem left open following our research pertains to other compression algorithms. Specifically, how can other compression mechanisms be adapted to deal with substring compression. This problem was presented in the original paper dealing with this problem [8], and we leave it open following our research as well. It seems as though algorithms such as Arithmetic encoding and algorithms relying on the Burrows–Wheeler transform would be more difficult to adapt to substring compression if at all possible, however this is simply a conjecture and such research should be of independent interest.
2. Another interesting problem may be to investigate the solutions for substrings for other classic stringology problems. For example, in the classical edit-distance problem, how can we efficiently retrieve the edit distance for any two substrings of the strings at hand. Specifically, can the known solutions inherently solve the problem for substrings as well? Or how can they be adapted to work for substrings?

References

[1] S. Alstrup, G.S. Brodal, T. Rauhe, New data structures for orthogonal range searching, in: IEEE Symposium on Foundations of Computer Science, 2000, pp. 198–207.
[2] A. Amir, D. Keselman, G.M. Landau, M. Lewenstein, N. Lewenstein, M. Rodeh, Text indexing and dictionary matching with one error, J. Algorithms 37 (2000) 309–325.
[3] M.A. Bender, M. Farach-Colton, The level ancestor problem simplified, Theor. Comput. Sci. 321 (2004) 5–12.
[4] P. Bille, I.L. Gørtz, Substring range reporting, in: Proc. of Symposium on Combinatorial Pattern Matching (CPM), 2011, pp. 299–308.
[5] T.M. Chan, K.G. Larsen, M. Patrascu, Orthogonal range searching on the RAM, revisited, in: Symposium on Computational Geometry, 2011, pp. 1–10.
[6] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Rasala, A. Sahai, A. Shelat, Approximating the smallest grammar: Kolmogorov complexity in natural models, in: J.H. Reif (Ed.), STOC, ACM, 2002, pp. 792–801.
[7] X. Chen, S. Kwong, M. Li, A compression algorithm for DNA sequences and its applications in genome comparison, in: RECOMB, 2000, p. 107.
[8] G. Cormode, S. Muthukrishnan, Substring compression problems, in: SODA'05: Proceedings of the Sixteenth Annual ACM–SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2005, pp. 321–330.
[9] A.J. Cox, T. Jakobi, G. Rosone, O. Schulz-Trieglaff, Comparing DNA sequence collections by direct comparison of compressed text indexes, in: B.J. Raphael, J. Tang (Eds.), WABI, in: Lect. Notes Comput. Sci., vol. 7534, Springer, 2012, pp. 214–224.
[10] M. Crochemore, L. Ilie, Computing longest previous factor in linear time and applications, Inf. Process. Lett. 106 (2008) 75–80.
[11] M. Crochemore, L. Ilie, W.F. Smyth, A simple algorithm for computing the Lempel–Ziv factorization, in: DCC, IEEE Computer Society, 2008, pp. 482–488.
[12] M. Crochemore, C.S. Iliopoulos, M. Kubica, M.S. Rahman, T. Walen, Improved algorithms for the range next value problem and applications, in: STACS 2008, 25th Annual Symposium on Theoretical Aspects of Computer Science, in: LIPIcs, vol. 1, Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, Germany, 2008, pp. 205–216.
[13] M. Farach, Optimal suffix tree construction with large alphabets, in: FOCS'97: Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS'97), IEEE Computer Society, Washington, DC, USA, 1997, p. 137.
[14] P. Ferragina, Dynamic text indexing under string updates, J. Algorithms 22 (1997) 296–328.
[15] P. Ferragina, S. Muthukrishnan, M. de Berg, Multi-method dispatching: A geometric approach with applications to string matching problems, in: STOC, 1999, pp. 483–491.
[16] K. Goto, H. Bannai, Simpler and faster Lempel–Ziv factorization, CoRR (2012), arXiv:1211.3642 [cs.DS].
[17] D. Harel, R.E. Tarjan, Fast algorithms for finding nearest common ancestors, SIAM J. Comput. 13 (1984) 338–355.
[18] D. Hermelin, S. Landau, G. Landau, O. Weimann, A unified algorithm for accelerating edit-distance via text-compression, in: Proc. of the 26th International Symposium on Theoretical Aspects of Computer Science (STACS), 2009, pp. 529–540.
[19] W.K. Hon, R. Shah, S.V. Thankachan, J.S. Vitter, On position restricted substring searching in succinct space, J. Discrete Algorithms 17 (2012) 109–114.
[20] J. Kärkkäinen, D. Kempa, S.J. Puglisi, Linear time Lempel–Ziv factorization: Simple, fast, small, in: J. Fischer, P. Sanders (Eds.), CPM, in: Lect. Notes Comput. Sci., vol. 7922, Springer, 2013, pp. 189–200.
[21] O. Keller, T. Kopelowitz, S. Landau, M. Lewenstein, Generalized substring compression, in: Combinatorial Pattern Matching, 20th Annual Symposium, CPM 2009, in: Lect. Notes Comput. Sci., vol. 5577, Springer, 2009, pp. 26–38.
[22] O. Keller, T. Kopelowitz, M. Lewenstein, Range non-overlapping indexing and successive list indexing, in: WADS'07: Proceedings of the 10th International Workshop on Algorithms and Data Structures, in: Lect. Notes Comput. Sci., vol. 4619, Springer-Verlag, 2007, pp. 626–631.
[23] T. Kopelowitz, M. Lewenstein, E. Porat, Persistency in suffix trees with applications to string interval problems, in: Proc. of Symposium on String Processing and Information Retrieval (SPIRE), 2011, pp. 67–80.
[24] S. Kuruppu, S.J. Puglisi, J. Zobel, Relative Lempel–Ziv compression of genomes for large-scale storage and retrieval, in: E. Chávez, S. Lonardi (Eds.), SPIRE, in: Lect. Notes Comput. Sci., vol. 6393, Springer, 2010, pp. 201–206.
[25] H.P. Lenhof, M. Smid, Using persistent data structures for adding range restrictions to searching problems, RAIRO Theor. Inform. Appl. 28 (1994) 25–49.
[26] M. Lewenstein, Orthogonal range searching for text indexing, CoRR (2013), arXiv:1306.0615 [cs.DS].
[27] V. Mäkinen, G. Navarro, Position-restricted substring searching, in: LATIN 2006: Theoretical Informatics, 7th Latin American Symposium, in: Lect. Notes Comput. Sci., vol. 3887, Springer, 2006, pp. 703–714.
[28] E.M. McCreight, A space-economical suffix tree construction algorithm, J. ACM 23 (1976) 262–272.
[29] S. Muthukrishnan, Personal communication with the second author, 2005.
[30] Y. Nekrich, G. Navarro, Sorted range reporting, in: F.V. Fomin, P. Kaski (Eds.), SWAT, in: Lect. Notes Comput. Sci., vol. 7357, Springer, 2012, pp. 271–282.
[31] E. Rivals, O. Delgrange, J.P. Delahaye, M. Dauchet, M.O. Delorme, A. Hénaut, E. Ollivier, Detection of significant patterns by compression algorithms: the case of approximate tandem repeats in DNA sequences, Comput. Appl. Biosci. 13 (1997) 131–136.

[32] W. Rytter, Application of Lempel–Ziv factorization to the approximation of grammar-based compression, Theor. Comput. Sci. 302 (2003) 211–222.
[33] E. Ukkonen, On-line construction of suffix trees, Algorithmica 14 (1995) 249–260.
[34] P. Weiner, Linear pattern matching algorithms, in: 14th Annual Symposium on Switching and Automata Theory, IEEE, 1973, pp. 1–11.
[35] J. Ziv, A. Lempel, A universal algorithm for sequential data compression, IEEE Trans. Inf. Theory 23 (1977) 337–343.