Lempel-Ziv Index for q-Grams*

Juha Kärkkäinen and Erkki Sutinen

Department of Computer Science
P.O. Box 26 (Teollisuuskatu 23)
FIN-00014 University of Helsinki, Finland
{*Juha.Karkkainen,Erkki.Sutinen*}*@cs.Helsinki.FI*

September 24, 1996

Abstract

We present a new sublinear-size index structure for finding all occurrences of a given q-gram in a text. Such a q-gram index is needed in many approximate pattern matching algorithms. All earlier q-gram indexes require at least $O(n)$ space, where n is the length of the text. The new Lempel-Ziv index needs only $O(n/\log n)$ space provided q does not grow too fast with n. The new method takes advantage of repetitions in the text found by Lempel-Ziv parsing.

1 Introduction

The *approximate pattern matching problem* is as follows. Given *text* $\mathcal{T} = \mathcal{T}[1,n]$ and *pattern* $\mathcal{P} = \mathcal{P}[1,m]$ in alphabet Σ and an integer k, find all the text positions i such that an approximate occurrence of $\mathcal{P}$ with at most k differences ends at i. The difference between two strings α and β is measured as *edit distance* d: $d(\alpha, \beta)$ is the minimum number of *edit operations* (insertions, deletions, or changes) needed to convert α to β (or, equivalently, β to α). Approximate pattern matching has several applications, for example in textual databases, biocomputing, data communications, and data mining.

Filtration with q-grams. The basic solution for the approximate pattern matching problem is based on dynamic programming [10, 20]. It has $O(kn)$ time complexity which is often too slow for long texts. Therefore, several *filtration* schemes have been introduced in order to extract the text areas with potential matches, to be inspected by dynamic programming in the second phase of the algorithm. Each filtration scheme is based on its specific *filtration condition*, stating a necessary condition for an approximate match.

*This work was supported by the Academy of Finland.

There are several possible ways to express the filtration condition. Potential approximate matches can be recognized by character distributions [6], locating matching blocks of the pattern in the text [23], retrieving maximal matching substrings of the pattern in the text [4, 21] or even applying the Boyer-Moore idea of exact pattern matching [2] to the approximate pattern matching [19].

A very popylar family of filtration schemes, called the *q-family*, states the filtration condition in terms of *q-grams*. By a q-gram, we refer to any (sub)string of q characters. For example, pattern `MOROGORO` has 3-grams `MOR`, `ORO`, `ROG`, `OGO`, and `GOR`. An approximate match resembles the original pattern. Therefore, any approximate match must contain at least some common q-grams with the searched pattern. Thus, the applicability of q-grams in a filtration condition is intuitively clear.

Each member of the q-family is based on its characteristic filtration condition. However, common to all of them is that one must find the text q-grams that match the pattern q-grams. This can be done in two different ways. The *dynamic* approach scans the text in linear time, while the *static* approach utilizes a *q-gram index*. The static approach is superior when the same q-gram index can be used several times. The classical paper by Jokinen and Ukkonen [8], introducing the use of q-grams in pattern matching, presented a static algorithm. Among the last members in the q-family, we name the dynamic approaches by Ukkonen [21], Chang & Marr [5], Takaoka [18], and Sutinen & Tarhio [16, 17]. Static algorithms have been introduced by Myers [14], Holsti & Sutinen [7], and Sutinen & Tarhio [17]. Furthermore, the FLASH system [3], designed for approximate pattern matching in biological data, applies q-grams in a probabilistic algorithm. In addition, q-grams have been applied in information retrieval schemes [1, 11].

Current implementations of q-gram indexes. There are several potential implementations for the q-gram index needed in the static q-gram methods. The basic implementation is a pointer array of size $|\Sigma|^q$. Each entry of the array refers to the list of occurrences of the corresponding q-gram in the text. The index can be built in time $O(n + |\Sigma|^q)$ [8].

The problem with this approach is the exponential dependence of the size of the array on q. This results in unnecessarily large indexes, especially when only a small fraction of potential q-grams occurs in the text. This is true, e.g., for natural language texts and all cases where $n \ll |\Sigma|^q$.

A natural improvement of the indexing scheme stated above is to reduce the size of the pointer array by hashing. By a suitable selection of the hash function, the size and construction time are reduced to $O(n)$ without significant slowdown in searching.

A different solution to the indexing problem uses a trie structure. The text q-grams are stored into a trie structure, with each leaf storing the list of occurrences of the corresponding q-gram in the text (Figure 1). The trie can be built in O(n) time, for example by using an algorithm similar to creating a suffix tree [22].

All the above implementations can find all occurrences of a given q-gram in optimal $O(q + l)$ time, where l is the number of occurrences found. However, the implementations also suffer from the same drawback: because of the at least linear space consumption, the size of the index gets unpractically large with an increasing text length

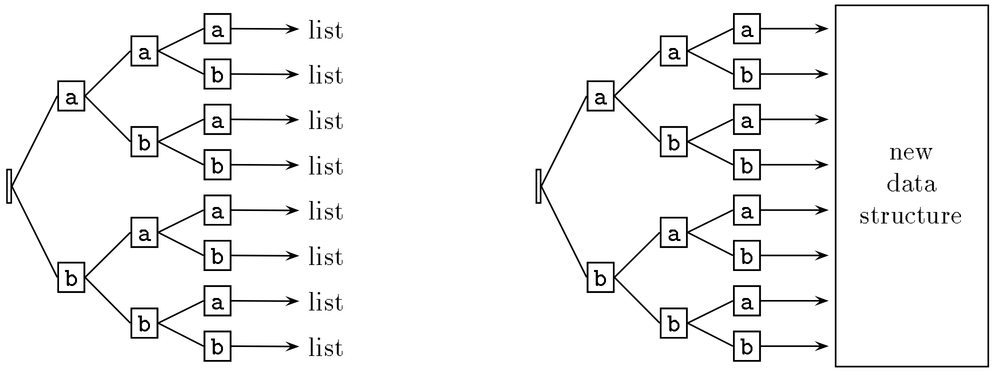

Figure 1: Comparison of trie implementations of a traditional 3-gram index and the new Lempel-Ziv index for 3-grams.

n. The hashing and trie structures can actually be implemented in $O(M)$ space, where M is the number of distinct q-grams of the text. The $O(n)$ space is required for the lists of occurrences.

Sketch of the new q-gram index. In the new Lempel-Ziv (LZ) index for q-grams, the lists of occurrences are replaced with a data structure that provides the same information in a more compact form (Figure 1). The (hashing or trie) structure for the distinct q-grams, now called the *primary index*, is still needed for providing a starting point for the search.

The compact representation of the occurrence lists takes advantage of repetitions in the text. With a repetition we mean a string that occurs (at least) twice in the text. The first occurrence is called the definition and the later occurrence is called the phrase. Every occurrence of a q-gram is either the first of its kind or a part of some phrase. The first occurrences can be stored in the primary index, e.g. in the leaves of a trie. All other occurrences are found using information about repetitions. Space is saved because a long repetition can represent several q-grams.

As an example, consider Figure 2. The first occurrence of the q-gram $\mathcal{Q}$ is found using the primary index. This phase of the algorithm is called the *primary search*. Next, we find the two definitions that contain the first occurrence. Just by using the information about repetitions, we can then deduce the locations of the second and fourth occurrence. This is called the *secondary search*. In the *tertiary search*, we find the definition containing the second occurrence which then leads to finding the third occurrence.

The tertiary search is applied for every new q-gram occurrence found. The difference between the secondary search and the tertiary search is that we can have precomputed information associated with the first occurrence to provide a starting point for the secondary search. The starting point for a tertiary search is the phrase that contains the current occurrence. However, this starting point is common to all the q-grams within that phrase, which makes searching a little more difficult.

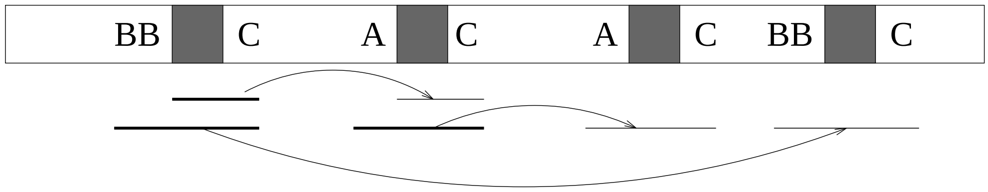

Figure 2: *Idea of the new method. The grey rectangles show the locations of a q-gram Q. A pair of lines, connected by an arrow, represents two occurrences of the same string (a repetition). The thick line is the definition and the thin line is the phrase.*

The repetitions in the text are found by a variation of Lempel-Ziv parsing. Lempel-Ziv parsing is also used in Ziv-Lempel text compression methods, where phrases are replaced with pointers to their definitions [24]. Our use of Lempel-Ziv parsing could be described as compressing the lists of occurrences instead of the text. Similar ideas, leading to quite different data structures, were used in [9] for exact pattern matching.

The LZ index can find all occurrences of a given q-gram in $O(q+l)$ time, the same as traditional q-gram indexes. The size of the LZ index is $O(M+N)$, where M is the number of distinct q-grams in the text and N is the number of repetitions needed to cover all q-grams except the first occurrence of each distinct q-gram. Obviously, $M+N$ is at most n, but it can be significantly less. In Section 8, we give conditions for the size to be sublinear. For example, if q is constant or grows slow enough with n, then $M+N = O(n/\log n)$. The size also gets smaller when the repetitiveness (compressibility) of the text increases.

In the next section, we will describe Lempel-Ziv parsing. Then, in Section 3, we give an outline of the search algorithm. The new data structures are based on certain combinatorial properties of intervals. These are studied in Section 4. The actual data structures are presented in Section 5 for the secondary search and in Section 6 for the tertiary search. The construction of the data structures is described in Section 7. In Section 8, we give theoretical and experimental results on the size of the new index, and then, in Section 9, compare the new scheme with traditional approaches. Finally, in Section 10, we point some aspects needing further research.

2 Lempel-Ziv Parsing

Let the string $\mathcal{T} = t_1t_2\ldots t_n$ be the text. We will use $\mathcal{T}[i,j]$ to denote the substring $t_it_{i+1}\ldots t_j$ of $\mathcal{T}$. The notation $[i,j]$ denotes an *interval* and can, in this paper, be interpreted to refer to the substring $t_it_{i+1}\ldots t_j$ with its content unknown or irrelevant. That is, $[i,j]$ denotes the *position* rather than the content. An alternative notation that is used frequently in this paper is $[i \to l] = [i, i+l-1]$.

The new index utilizes repetitions in the text to reduce the space requirement of the q-gram index. The information about the repetitions is represented by an LZ parse.

Definition 1 *Lempel-Ziv (LZ) parse* for a text $\mathcal{T}$ is a set of triplets (I_i, J_i, L_i), $i = 1, \ldots, N$, where $I_i > J_i$ and $\mathcal{T}[I_i \to L_i] = \mathcal{T}[J_i \to L_i]$.

The intervals $[I_i \to L_i]$ are called *phrases* and $[J_i \to L_i]$ are called *definitions*. They are intervals instead of substrings because all that the searching algorithm has to know about their content is that, for all i, $[I_i \to L_i]$ and $[J_i \to L_i]$ have the same content.

LZ parsing was originally used for measuring the complexity of a text [12]. LZ parsing is also utilized in Ziv-Lempel compression, where the contents of phrases are replaced with references to definitions [24]. Our use of the LZ parse utilizes the identity $\mathcal{T}[I_i \to L_i] = \mathcal{T}[J_i \to L_i]$ in a different way.

If there is an occurrence of a q-gram $\mathcal{Q}$ inside the phrase $[I_i \to L_i]$, there is then an occurrence of $\mathcal{Q}$ within the definition $[J_i \to L_i]$, too. The two occurrences are called the *phrase occurrence* and its *definition*, respectively. In the LZ index, phrase occurrences are found through their definitions. A definition of a phrase occurrence may itself be a phrase occurrence and have its own definition. Circular definitions are not possible, however, due to the condition $I_i > J_i$.

Definition 1 does not specify how to choose the phrases and definitions. In the original Lempel-Ziv parsing, the phrases are formed starting from the beginning of the text so that each phrase begins where the previous phrase ends[1] and is as long as Definition 1 allows. The idea is to cover as much of the text as possible with as few phrases as possible.

Our goal is to cover as many q-grams as possible with as few phrases as possible. The phrases are computed in the same way as in the original LZ parsing except consecutive phrases overlap by $q - 1$ characters. In detail, the method is described in Algorithm 1. For $q = 0$, Algorithm 1 computes exactly the original LZ parse.

Algorithm 1: LZParse
Input: text $\mathcal{T}$ of length n
Output: LZ parse of text $\mathcal{T}$, starting points of all non-covered q-grams

$i \leftarrow j \leftarrow I_1 \leftarrow 1$
while $I_i \leq n - q + 1$ **do**
  $L_i \leftarrow \max\{L \mid \exists J < I_i : \mathcal{T}[J \to L] = \mathcal{T}[I_i \to L]\}$
  if $L_i < q$ **then** % non-covered q-gram
    $P_j \leftarrow I_i$
    $j \leftarrow j + 1$
    $I_i \leftarrow I_i + 1$
  else % phrase
    $J_i \leftarrow \min\{J \mid \mathcal{T}[J \to L_i] = \mathcal{T}[I_i \to L_i]\}$
    $I_{i+1} \leftarrow I_i + L_i - q + 1$
    $i \leftarrow i + 1$
$N \leftarrow i - 1$
$M \leftarrow j - 1$
return $\{(I_1, J_1, L_1), \ldots, (I_N, J_N, L_N)\}, \{P_1, \ldots, P_M\}$

---

[1] To be exact, one character is left between consecutive phrases.

The key properties of the resulting LZ parse are described below.

Fact 2 *The LZ parse of a text $\mathcal{T}$ constructed by Algorithm 1 has the following properties.*

(i) All phrases are at least q characters long and two phrases overlap by at most $q-1$ positions.

(ii) A definition $[J_i \to L_i]$ is the first occurrence of the string $\mathcal{T}[J_i \to L_i]$ in $\mathcal{T}$.

(iii) A phrase $[I_i \to L_i]$ is not *the first occurrence of the string $\mathcal{T}[I_i \to L_i]$ in $\mathcal{T}$.*

As a consequence of the last two items, we get the following corollary, which has key importance to the tertiary search described in Section 6.

Corollary 3 *No phrase contains a definition.*

The algorithm also computes the starting positions $P_1, \ldots, P_M$ of all non-covered q-gram occurrences. By Fact 2(iii), the first occurrence of each q-gram cannot be covered by any phrase. All other q-grams, however, are covered as shown by the following lemma.

Lemma 4 *The q-gram occurrences with starting positions $P_1, \ldots, P_M$ returned by Algorithm 1 are exactly the first occurrences of the distinct q-grams appearing in $\mathcal{T}$.*

Proof. The occurrences are clearly the non-covered occurrences, i.e., the occurrences not contained by any phrase. By Fact 2(iii), all first q-gram occurrences must then be among them. We still need to show that every occurrence $[P_j \to q]$ is a first occurrence. An occurrence $[P_j \to q]$ is formed in the algorithm only when the attempt to form a phrase $[I_i \to L_i]$, where $I_i = P_j$, fails because $L_i < q$. This means that $[P_j \to q]$ is the first occurrence of the q-gram $\mathcal{T}[P_j \to q]$. □

An example of the operation of Algorithm 1 is shown in Figure 3.

Algorithm 1 clearly works in $O(N+M)$ time and space except possibly the computation of L_i and J_i on lines 3 and 9. To compute L_i and J_i, the algorithm builds a suffix tree of the text. The values L_i and J_i are naturally found when the suffix starting at I_i is added to the suffix tree. Using the suffix tree construction algorithms by McCreight [13] or by Ukkonen [22], the algorithm then executes in $O(n)$ time and space.

3 Outline of the Search Algorithm

With the new index, the phrase occurrences are found through their definitions. The resulting basic search strategy is the following.

1. [Primary search] Find the first occurrence $[p \to q]$ of the q-gram $\mathcal{Q}$ in $\mathcal{T}$.
2. [Secondary search] Find the definitions $[J_i \to L_i]$ that contain $[p \to q]$. Then there is a phrase occurrence within each corresponding phrase $[I_i \to L_i]$.

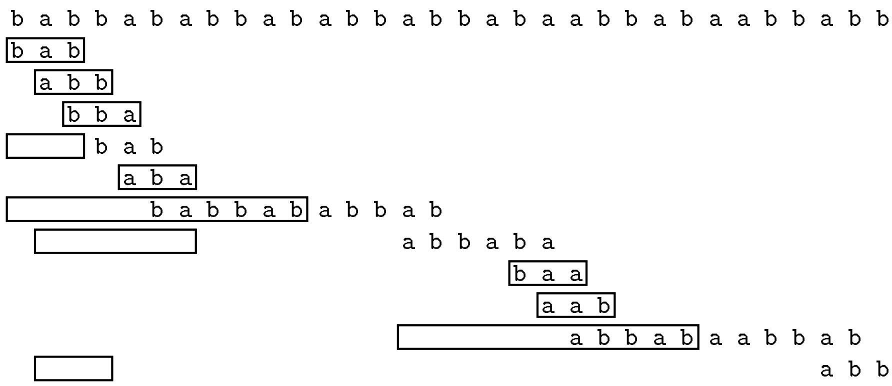

Figure 3: An example of LZ parsing by Algorithm 1 for $q = 3$. The text is on the first line. The rest of the lines show the phrases and the non-covered 3-grams. Each box marks the position of the first occurrence of the string on that line. In the case of a 3-gram, the string itself is boxed. For a phrase, the box shows the position of the corresponding definition.

3. [Tertiary search] For each phrase occurrence found in either step 2 or step 3, find the definitions $[J_i \to L_i]$ that contain it. Then there is an additional phrase occurrence within each corresponding phrase $[I_i \to L_i]$.

Note that to determine containment, one only needs to compare the end point; no character comparisons are needed. A more detailed description of the search is given in Algorithm 2.

In the algorithm, the function PrimarySearch($\mathcal{Q}$) returns j such that $[P_j \to q]$ is the position of the first occurrence of $\mathcal{Q}$. The primary search is implemented using the *primary index*, a traditional q-gram index constructed from the first occurrences computed by Algorithm 1. The primary index differs from an ordinary q-gram index only by the fact that there is only one occurrence of each distinct q-gram. A trie implementation, for example, can be built in $O(\min\{Mq, n\})$ time and $O(M)$ space, and can answer the queries in $O(q)$ time.

The function SecondarySearch(j) in Algorithm 2 computes all i such that the definition $[J_i \to L_i]$ contains $[P_j \to q]$. The implementation is described in Section 5. The function TertiarySearch(p, i) computes all j such that the definition $[J_j \to L_j]$ contains $[p \to q]$, when $[p \to q]$ is known to be contained by the phrase $[I_i \to L_i]$. The implementation is described in Section 6.

4 Intervals

The secondary and tertiary search problems above were stated in terms of definitions, phrases and q-grams occurrences, all of which are intervals. Our secondary and ter-

Algorithm 2: LZSEARCH
Input: q-gram $\mathcal{Q}$
Output: starting points of all occurrences of $\mathcal{Q}$

```
j ← PrimarySearch(Q)
T ← {P_j}
S ← ∅
R ← SecondarySearch(j)
for each i ∈ R do
    T ← T ∪ {I_i − J_i + P_j}
    S ← S ∪ {(I_i − J_i + P_j, i)}
while S ≠ ∅ do
    Let (p, i) ∈ S
    S ← S \ {(p, i)}
    R ← TertiarySearch(p, i)
    for each j ∈ R do
        T ← T ∪ {I_j − J_j + p}
        S ← S ∪ {(I_j − J_j + p, j)}
return T
```

tiary search algorithms are based on some combinatorial properties of intervals. In this section, we will describe these key properties.

We start by defining two partial orders on intervals.

Definition 5 An interval $[I, J]$ is *contained* by an interval $[I', J']$, denoted by $[I, J] \subseteq [I', J']$, iff $I' \leq I$ and $J \leq J'$. There are two degrees of inequality:

$$\begin{array}{lll} [I,J] \subset [I',J'] & \text{iff} & [I,J] \subseteq [I',J'] \text{ and } [I,J] \neq [I',J']. \\ [I,J] \prec [I',J'] & \text{iff} & [I,J] \subseteq [I',J'],\ I \neq I' \text{ and } J \neq J'. \end{array}$$

In the last case, we say that $[I, J]$ is *nested* within $[I', J']$.

Definition 6 An interval $[I, J]$ *precedes* an interval $[I', J']$, denoted by $[I, J] \leq [I', J']$, iff $I \leq I'$ and $J \leq J'$. There are two degrees of inequality:

$$\begin{array}{lll} [I,J] < [I',J'] & \text{iff} & [I,J] \leq [I',J'] \text{ and } [I,J] \neq [I',J']. \\ [I,J] \ll [I',J'] & \text{iff} & [I,J] \leq [I',J'],\ I \neq I' \text{ and } J \neq J'. \end{array}$$

Two intervals $[I, J]$ and $[I', J']$ are *comparable* in $\subseteq$ $(\leq)$ if $[I, J] \subseteq (\leq)[I', J']$ or $[I', J'] \subseteq (\leq)[I, J]$.

Example 1 $[3, 4] \prec [2, 5] \subset [2, 7]$, but $[1, 3]$ and $[2, 5]$ are incomparable in $\subseteq$. $[1, 3] \ll [2, 5] < [2, 7]$, but $[2, 5]$ and $[3, 4]$ are incomparable in $\leq$.

The two partial orders are related by the following lemma.

Lemma 7 (Orthogonality) *Two intervals* $[I, J]$ *and* $[I', J']$ *are always comparable in either* $\subseteq$ *or* $\leq$. *The intervals are comparable in both* $\subseteq$ *and* $\leq$ *iff* $I = I'$ *or* $J = J'$.

We say that $\subseteq$ and $\leq$ are *orthogonal*.

Corollary 8 *Two intervals* $[I, J]$ *and* $[I', J']$ *are incomparable in* $\leq$ *iff* $[I, J] \prec [I', J']$ *or* $[I', J'] \prec [I, J]$.

The basic idea of our algorithms for secondary and tertiary search is to sort the definitions by $\leq$. The order $\leq$ is a partial order, but Corollary 8 allows us to partition the set of definitions into sets for which $\leq$ is a total order.

Definition 9 Let $\mathcal{S}$ be a set of intervals. The *nesting level* of an interval $[I, J]$ in $\mathcal{S}$, denoted by $nl_{\mathcal{S}}([I, J])$, is the largest integer k for which there exists a set $\{[I_1, J_1], \ldots, [I_k, J_k]\} \subset \mathcal{S}$ such that $[I, J] \prec [I_1, J_1] \prec \cdots \prec [I_k, J_k]$.

Let $NL_k(\mathcal{S})$ denote the set of intervals of $\mathcal{S}$ with nesting level k in $\mathcal{S}$, i.e., $NL_k(\mathcal{S}) = \{[I, J] \in \mathcal{S} \mid nl_{\mathcal{S}}([I, J]) = k\}$. Let H be the largest integer for which $NL_H(\mathcal{S})$ is nonempty. Then, $NL_0(\mathcal{S}), NL_1(\mathcal{S}), \ldots, NL_H(\mathcal{S})$ is called the *nesting level partition* of $\mathcal{S}$. The key property of nesting levels is the following.

Theorem 10 *Let* $\mathcal{S}$ *be a set of intervals and* k *a non-negative integer. Then,* $\leq$ *is a total order for* $NL_k(\mathcal{S})$.

Proof. Let $[I, J]$ and $[I', J']$ be intervals in $NL_k(\mathcal{S})$. Assume, contradictory to the theorem, that $[I, J]$ and $[I', J']$ are incomparable in $\leq$. Then, by Corollary 8, $[I, J] \prec [I', J']$ or $[I', J'] \prec [I, J]$. Assume $[I, J] \prec [I', J']$. By Definition 9, $[I', J'] \prec [I_1, J_1] \prec \cdots \prec [I_k, J_k]$ for some $\{[I_1, J_1], \ldots, [I_k, J_k]\} \subset \mathcal{S}$. Then $[I, J] \prec [I', J'] \prec [I_1, J_1] \prec \cdots \prec [I_k, J_k]$ and therefore $nl_{\mathcal{S}}([I, J]) > k$, i.e., $[I, J] \notin NL_k(\mathcal{S})$. $\square$

5 Secondary Search

The secondary search problem can be stated as follows.

> Given the first occurrence of the q-gram $\mathcal{Q}$ in the text $\mathcal{T}$ at position $[p \to q]$, find all definitions $[J_i \to L_i]$, $i = 1, \ldots, N$, that contain $[p \to q]$.

As can be seen from this formulation, the problem is concerned with intervals rather than strings. In fact, neither the text $\mathcal{T}$ nor the q-gram $\mathcal{Q}$ is needed at all.

The secondary search is implemented using a new data structure, called a *nesting leveled list* (NLL), that is based on the concepts introduced in Section 4. Let $\mathcal{D}$ denote the set of definitions and let $NL_0(\mathcal{D}), NL_1(\mathcal{D}), \ldots, NL_H(\mathcal{D})$ be its nesting level partition. The nesting leveled list of $\mathcal{D}$, NLL($\mathcal{D}$), consists of a separate $\leq$-ordered bidirectional list for each nesting level set $NL_k(\mathcal{D})$, $k = 0, \ldots, H$. Recall that $\leq$ is a total order for $NL_k(\mathcal{D})$. There are also links between the nesting levels, called *parent links*, from every definition to one of its parents. A *parent* of a definition $[J_i \to L_i] \in NL_k(\mathcal{D})$ is a definition $[J_j \to L_j] \in NL_{k-1}(\mathcal{D})$ such that $[J_i \to L_i] \prec [J_j \to L_j]$. Every definition $[J_i \to L_i]$ has one or more parents unless $[J_i \to L_i] \in NL_0(\mathcal{D})$.

Suppose we know that a definition $[J_i \to L_i] \in NL_k(\mathcal{D})$ contains $[p \to q]$. Then the definitions in $NL_k(\mathcal{D})$ containing $[p \to q]$ form a continuous neighborhood of $[J_i \to L_i]$ in the list for $NL_k(\mathcal{D})$ and they can be found quickly by following the bidirectional list. Furthermore, if $k > 0$, then every parent of $[J_i \to L_i]$ also contains $[p \to q]$. Therefore, using parent links and the nesting level lists, we can find all definitions in the nesting levels $0, \ldots, k$ that contain $[p \to q]$. The procedure is described in detail in the algorithm ExtendFind.

```
ExtendFind(i,p)          % [p → q] ⊆ [J_i → L_i]
    T ← ∅
    repeat
        T ← T ∪ {i}
        k ← Pred(i)
        while k ≠ 0 and [p → q] ⊆ [J_k → L_k] do
            T ← T ∪ {k}
            k ← Pred(k)
        k ← Succ(i)
        while k ≠ 0 and [p → q] ⊆ [J_k → L_k] do
            T ← T ∪ {k}
            k ← Succ(k)
        i ← Parent(i)
    until i = 0
    return T
```

In ExtendFind, NLL($\mathcal{D}$) is represented by the operations Pred, Succ and Parent defined as follows.

Definition 11 Let $[J_i \to L_i]$ be the immediate predecessor of $[J_j \to L_j]$ in the list for their common nesting level. Then $\text{Pred}(j) = i$ and $\text{Succ}(i) = j$. If $[J_i \to L_i]$ does not have a predecessor (successor), then $\text{Pred}(i) = 0$ ($\text{Succ}(i) = 0$).

Definition 12 Let $[J_i \to L_i] \in NL_k(\mathcal{D})$, $k > 0$. Then $\text{Parent}(i) = j$ for some j such that $[J_j \to L_j] \in NL_{k-1}(\mathcal{D})$ and $[J_i \to L_i] \prec [J_j \to L_j]$. If $[J_i \to L_i] \in NL_0(\mathcal{D})$, then $\text{Parent}(i) = 0$.

Given algorithm ExtendFind, the secondary search problem is easy to solve if we can find a definition that contains $[p \to q]$ in the deepest nesting level that has one. This we will do through precomputed def-values.

Definition 13 Let $[P_j \to q]$ be the first occurrence of a q-gram. Then, $\text{Def}(j) = i$ for some i such that $[P_j \to q] \subseteq [J_i \to L_i]$ and $[J_i \to L_i] \in NL_k(\mathcal{D})$, where k is the deepest level that has definitions containing $[P_j \to q]$. If no definition contains $[P_j \to q]$, then $\text{Def}(j) = 0$.

The algorithm SecondarySearch now solves the secondary search problem.

```
SecondarySearch(j)        % [P_j → q] is an occurrence
    i ← Def(j)
    if i = 0 then return ∅
    else return ExtendFind(i, P_j)
```

Each execution of a loop body in ExtendFind adds a new definition containing $[p \to q]$ into T and each of the definitions in T will lead to a discovery of a new occurrence. Therefore, the whole secondary search runs in linear time in the number of occurrences found. The size of the nesting leveled list is $O(N)$, and the Def pointers require $O(M)$ space. The construction of the data structures will be described in Section 7.

6 Tertiary Search

The tertiary search problem is very similar to the secondary search problem.

> Given an occurrence of the q-gram $\mathcal{Q}$ in the text $\mathcal{T}$ at a position $[p \to q]$ contained by a phrase $[I_i \to L_i]$, find all definitions $[J_j \to L_j]$, $j = 1, \ldots, N$, that contain $[p \to q]$.

There is two differences to the secondary search. First, the occurrence at $[p \to q]$ is not a first occurrences, and second, we know the phrase that contains the occurrence. The procedure ExtendFind can be used with the phrase occurrences as well as with the first occurrences. However, finding the starting point for ExtendFind is more difficult. Associating a definition with each phrase occurrence would defeat the purpose of saving space. Instead, we will find the starting point through the phrase that is known to contain the occurrence.

By Corollary 3, a phrase cannot contain any definitions. Therefore, a definition $[J_j \to L_j]$ can be classified as one of the following three types with respect to a phrase $[I_i \to L_i]$.

1. Containing definition: $[I_i \to L_i] \subset [J_j \to L_j]$.
2. Preceding definition: $[J_j \to L_j] \ll [I_i \to L_i]$.
3. Succeeding definition: $[I_i \to L_i] \ll [J_j \to L_j]$.

Every containing definition contains all occurrences $[p \to q]$ within $[I_i \to L_i]$. A preceding or succeeding definition may contain $[p \to q]$ if it overlaps $[I_i \to L_i]$ by at least q characters.

Let K_i be the deepest nesting level that has a definition containing $[I_i \to L_i]$ and let H_i be the deepest nesting level that has a preceding or succeeding definition with an overlap of at least q. Then all nesting levels $0, \ldots, K_i$ have a definition that contains $[p \to q] \subseteq [I_i \to L_i]$, and the starting point for ExtendFind must be between levels K_i and H_i, inclusive.

The *preceding edge* for a phrase $[I_i \to L_i]$ contains the last preceding definition from each of the levels $K_i+1, \ldots, H_i$. Similarly, the *succeeding edge* contains the first succeeding definition from each of the levels $K_i+1, \ldots, H_i$. As levels $K_i+1, \ldots, H_i$ have no containing definitions, the last preceding definition and the first succeeding definition on each level are adjacent in their nesting level list. If neither of the two contains $[p \to q]$, then neither does any other definition on the same level. Therefore, it is enough to search the preceding and succeeding edges for the starting point for ExtendFind.

The tertiary search is described in detail in the algorithm TertiarySearch. The algorithm uses Bottom and PrecEdge which are precomputed values defined as follows.

Definition 14 Let $[I_i \to L_i]$ be a phrase. Then $\textsc{Bottom}(i) = j$ for some j such that $[J_j \to L_j] \in NL_{K_i}(\mathcal{D})$ contains $[I_i \to L_i]$. For $1 \le l \le H_i - K_i$, $\textsc{PrecEdge}(i,l) = j$ such that $[J_j \to L_j]$ is the last definition in $NL_{K_i+l}(\mathcal{D})$ that precedes $[I_i \to L_i]$.

If necessary, we will add a dummy definition (e.g. $[0,0]$) at the start of every nesting level list such that $\textsc{PrecEdge}(i,l)$ is defined for every $i = 1, \ldots, N$ and every $l = 1, \ldots, H_i - K_i$. To simplify the algorithm, we also define $\textsc{PrecEdge}(i, H_i - K_i + 1) = 0$.

TertiarySearch(p,i) % $[p \to q] \subseteq [I_i \to L_i]$

  $j \leftarrow \textsc{Bottom}(i)$
  $l \leftarrow 1$
  $k \leftarrow \textsc{PrecEdge}(i, 1)$
  while $k \neq 0$ **and** $[p \to q] \subseteq [J_k \to L_k]$ **or**
    $\textsc{Succ}(k) \neq 0$ **and** $[p \to q] \subseteq [J_{\textsc{Succ}(k)} \to L_{\textsc{succ}(k)}]$ **do**
    $j \leftarrow k$
    $l \leftarrow l + 1$
    $k \leftarrow \textsc{PrecEdge}(i, l)$
  if $j = 0$ **then return** $\emptyset$
  else
    if $[p \to q] \not\subseteq [J_j \to L_j]$ **then** $j \leftarrow \textsc{Succ}(j)$
    return $\textsc{ExtendFind}(j, p)$

Each round of the **while** loop in TertiarySearch passes a definition that contains $[p \to q]$. Each of those definitions will be found and returned by ExtendFind and will lead to a discovery of a new occurrence. Therefore, the whole tertiary search runs in time linear in the number of occurrences found.

The space requirement for the Bottom pointers is clearly $O(N)$. For each value $\textsc{PrecEdge}(i,l)$, there is at least one preceding or succeeding definition that overlaps $[I_i \to L_i]$ by at least q positions. By Fact 2(i), phrases can not overlap each other by more than $q-1$ positions. Thus, a definition may overlap at most two phrases by q positions without containing them. Therefore, the total space requirement of the PrecEdge values is $O(N)$.

7 Construction

The construction of the LZ index starts by LZ parsing the text using Algorithm LZPARSE. Then, the primary index and the data structures used in the secondary and tertiary search are built from the LZ parse and the first occurrences of q-grams. We will next sketch an algorithm that constructs all the secondary and tertiary search structures: the nesting leveled list, and the DEF, BOTTOM and PRECEDGE pointers.

The LZ parsing outputs three kinds of intervals: phrases, definitions and q-gram occurrences. The secondary and tertiary index construction starts by sorting all these intervals together by the starting position. The intervals are then processed in the sorted order:

- When a definition is encountered, it is added to the (originally empty) nesting leveled list NLL.
- When an occurrence $[P_i \to q]$ is encountered, the value $\text{DEF}(i)$ is computed.
- When a phrase $[I_i \to L_i]$ is encountered, $\text{BOTTOM}(i)$ and $\text{PRECEDGE}(i, l)$, for $1 \leq l \leq H_i - K_i$, are computed.

A more detailed description of the construction algorithm is given below.

1. All three kinds of intervals are sorted together into increasing order of the starting positions. Among intervals with the same starting position, definitions are placed before phrases and occurrences, and the length is used as an additional key. The sorting can be done in $O(n)$ time and space using radix sorting.

2. The intervals are processed in the sorted order. Let $[I, J]$ be the interval to be processed next and let K be the deepest level in the current NLL that has a definition $[J_j \to L_j]$ containing $[I, J]$. The level K and the definition $[J_j \to L_j]$ are found by a binary search among the latest definitions of the nesting levels. Over the whole algorithm this takes $O(N \log H)$ time, where H is the depth of NLL.

3. The rest of the processing of the interval $[I, J]$ depends on its type.

   (a) Let $[I, J]$ be the definition $[J_i \to L_i]$. If $[J_i \to L_i] \prec [J_j \to L_j]$, then add $[J_i \to L_i]$ into NLL as the last definition on level $K + 1$ and set $\text{PARENT}(i) = j$. Otherwise (if $J_i + L_i = J_j + L_j$) add $[J_i \to L_i]$ into NLL as the last definition on level K and set $\text{PARENT}(i) = \text{PARENT}(j)$.

   In the former case, $[J_i \to L_i]$ may belong to the succeeding edge of the last phrase $[I_k \to L_k]$. Following Section 6, let K_k and H_k be such that the current PRECEDGE list for $[I_k \to L_k]$ is from level $K_k + 1$ to H_k. If $H_k = K$ and $[J_i \to L_i]$ overlaps $[I_k \to L_k]$ by at least q characters, then set $\text{PRECEDGE}(k, K + 1 - K_k) = \text{PREC}(i)$. If $\text{PREC}(i) = 0$ ($[J_i \to L_i]$ is the first definition on level $K + 1$), add a dummy definition at the start of level $K + 1$.

   (b) If $[I, J]$ is the occurrence $[P_i \to q]$, set $\text{DEF}(i) = j$.

(c) If $[I, J]$ is the phrase $[I_i \to L_i]$, set $\textsc{Bottom}(i) = j$. Additionally, for $l = 1, 2, \ldots$ set $\textsc{PrecEdge}(i, l) = k$, where $[J_k \to L_k]$ is the currently last definition on level $K + l$, until $[J_k \to L_k]$ no more overlaps $[I_i \to L_i]$ by at least q positions.

In total, the construction works in $O(n + N \log H)$ time and $O(n)$ space.

8 The Size of LZ Index

The primary index of our method can be implemented in $O(M)$ space, where M is the number of distinct q-grams in the text. The secondary and tertiary indexes require $O(N)$ space, where N is the number of phrases in the LZ parse of the text. The total space requirement is therefore $O(M + N)$. In the following, we show that this can be sublinear (cf. [9]). All analysis below assumes that the size of the alphabet is constant.

Lemma 15 *The number of phrases of length at most $l \geq q$ is at most $(c^{l-q+1} - 1)M$, where c is the size of the alphabet.*

Proof. Consider the primary occurrences and the phrases of length at most l, all extended from the end up to length $l + 1$. Due to the properties of Algorithm 1, each such extended interval must be the first occurrence of the corresponding substring and, therefore, the extended intervals have all different contents. Each extended interval starts with one of the M distinct q-grams appearing in the text followed by $l - q + 1$ characters. Thus, there can be at most $c^{l-q+1}M$ extended intervals, of which M are extensions of primary occurrences. □

Lemma 16 *Let $k = \lceil \log_c((M + N)/M) \rceil$. Then the length of the text is*

$$n \geq k(M + N) - M \sum_{i=0}^{k-1} c^i + M + q - 1 > \left(k - \frac{c}{c-1}\right)(M + N) + M + q - 1.$$

Proof. If there are no phrases ($N = 0$, $k = 0$), $n = M + q - 1$. Each phrase overlaps the preceding phrase or first q-gram occurrence by $q - 1$ characters and, therefore, a phrase of length l adds $l - q + 1$ to the length of the text. Lemma 15 limits the number of short phrases. Specifically, with k such that $(c^{k-1} - 1)M < N \leq (c^k - 1)M$, there must be phrases of length at least $k + q - 1$. From further application of Lemma 15, we can then deduce that N phrases add at least $kN - M \sum_{i=0}^{k-1}(c^i - 1)$ to the length of the text. Therefore,

$$n \geq M + q - 1 + kN - M \sum_{i=0}^{k-1}(c^i - 1) = k(M + N) - M \sum_{i=0}^{k-1} c^i + M + q - 1.$$

The result follows when we note that

$$\sum_{i=0}^{k-1} c^i = \frac{c^k - 1}{c - 1} < \frac{c}{c - 1} \frac{M + N}{M}.$$

□

Theorem 17 *If $M = O(n/(f(n)\log f(n)))$, for $f(n)$ such that $f(n)\log f(n) = O(n)$ and $f(n) = \Omega(1)$, then $N = O(n/\log f(n))$.*

Proof. If $N = \Omega(n/\log f(n))$, then $k = \Omega(\log f(n))$ and, by Lemma 16,

$$N < \frac{n - M - q + 1}{k - \frac{c}{c-1}} - M = O\left(\frac{n}{\log f(n)}\right).$$

□

Corollary 18 *If $M = O(n^{1-\epsilon})$, where ϵ is a positive constant, then $N = O(n/\log n)$.*

Proof. Choose $f(n) = n^{\epsilon}/\log n$ in Theorem 17. □

Corollary 19 *If $q \leq (1-\epsilon)\log_{|\Sigma|} n + C$, where ϵ is a positive constant and C is an arbitrary constant, then $M + N = O(n/\log n)$.*

The sublinearity conditions of Theorem 17, Corollary 18 and Corollary 19 are sufficient but not necessary. For a compressible text, the LZ index can have sublinear size, even if the conditions are not satisfied.

As is usual, the sizes refer to the number of words needed to store the data structure given the assumption that a word can store an arbitrary pointer. However, a pointer to the text, for example, needs at least $\log n$ bits. Thus, measured in bits, bytes or other constant units, the size of the LZ index under the conditions of Corollary 18 is actually $O(n)$ and the size of a traditional q-gram index is $O(n\log n)$. On the other hand, the text itself could be packed in $O(n/\log n)$ words of size $O(\log n)$. Therefore, let $||\mathcal{T}||$ denote the *size* (in contrast to the length) of the text in whichever space units are being used. Then, the size of the LZ index is $O(||\mathcal{T}||)$, while the size of a traditional q-gram index is $O(||\mathcal{T}||\log||\mathcal{T}||)$.

Table 1 shows measured values of M and N, and an estimated size of the LZ index for some texts. The actual size of the data structures depends on the implementation and may also depend on other statistics besides M and N (e.g. the shape of the primary index trie and the number of PRECEDGE-values). However, the estimate $16M + 24N$ bytes used in Table 1 should be fairly close to an average size of a reasonable implementation. Table 2 shows the estimated size as the text length increases.

9 Comparison to Traditional q-gram Indexes

As was illustrated in Figure 1, both a traditional q-gram index and the new LZ index need a primary index, a data structure to access information associated with a given q-gram. The rest of a traditional index consists of the lists of occurrences for the distinct q-grams. The LZ index replaces these with the data structures for the secondary and tertiary search. Both data structures can list the occurrences of a given q-gram in linear time in the number of occurrences. In practice, the LZ index is slower by a small constant factor.

The major difference between the old and the new methods is their space requirement. A traditional q-gram index requires $\Theta(n)$ space, independent of any other

Table 1: Measured values of M, the number of distinct q-grams, N, the number of phrases, and the estimated size $16M+24N$ bytes of the LZ index given in bytes per text character. The texts are English language, DNA sequence data and C programming language source code, all of length $1,000,000$.

	english			DNA			C code		
q	M	N	size	M	N	size	M	N	size
1	83	89520	2.15	4	105385	2.53	96	72274	1.74
2	2519	107904	2.63	16	118295	2.84	4079	87727	2.17
3	20613	133346	3.53	64	134596	3.23	26086	99038	2.79
4	71355	133466	4.34	256	156459	3.76	67016	95644	3.37
5	142062	116685	5.07	1024	186948	4.50	114059	88354	3.95
6	212031	101256	5.82	4096	231661	5.63	163367	79758	4.53
7	276814	87407	6.53	16384	297634	7.41	211269	71387	5.09
8	335910	75462	7.19	65019	381010	10.19	256440	63723	5.63
9	388282	65119	7.78	224860	410764	13.46	298219	57005	6.14
10	435081	55830	8.30	519731	295176	15.40	336752	50995	6.61

Table 2: The estimated size of the LZ index in bytes per text character for English text of varying length.

	$n/1000$				
q	500	1000	1500	2000	2500
1	2.51	2.15	2.00	1.89	1.82
2	3.14	2.63	2.42	2.27	2.18
3	4.24	3.53	3.22	3.01	2.87
4	5.16	4.34	3.98	3.74	3.57
5	5.97	5.07	4.67	4.39	4.21
6	6.79	5.82	5.38	5.08	4.88
7	7.52	6.53	6.07	5.74	5.53
8	8.20	7.19	6.72	6.38	6.16
9	8.78	7.78	7.31	6.96	6.74
10	9.29	8.30	7.84	7.50	7.28

Table 3: The size of the LZ index relative to the size of a traditional q-gram index for English text of varying length. The relative size is estimated by the formula $(16M + 24N)/(16M + 4n)$.

	$n/1000$				
q	500	1000	1500	2000	2500
1	0.62	0.54	0.50	0.47	0.46
2	0.77	0.65	0.60	0.56	0.54
3	0.93	0.82	0.76	0.72	0.69
4	0.91	0.84	0.81	0.78	0.76
5	0.85	0.81	0.79	0.77	0.76
6	0.82	0.79	0.77	0.76	0.75
7	0.79	0.77	0.76	0.75	0.75
8	0.78	0.77	0.76	0.75	0.74
9	0.78	0.76	0.75	0.75	0.74
10	0.77	0.76	0.75	0.75	0.74

parameters, particularly independent of q. In fact, the $\Theta(n)$-size suffix tree of the text can be used as a q-gram index for any q. Note that any q-gram index, including the LZ index, can be used as a q'-gram index for $q' < q$, but usually not for $q' > q$, at least not efficiently.

The LZ index has size $O(n/\log n)$ provided that q does not grow too fast with n. The size depends heavily on q. The advantage of the dependence on q is that one can control the size of the index by choosing q. The preferable value of q depends on the particular q-gram method. Let us look at a couple of examples.

In the original q-gram method, the optimal value of q depends on the pattern $\mathcal{P}$, the error limit k and the type of the text, but not on the length of the text [8]. The variation described in [7] has the same property. Therefore, for these methods, LZ index is truly sublinear in the length of the text.

In a q-gram method by Myers, the optimal value of q, for a random text, is $\log_{|\Sigma|} n$ [14]. However, using $q = (1-\epsilon)\log_{|\Sigma|} n$ instead weakens the efficiency of the algorithm only slightly. Therefore, a sublinear-size LZ index is a possibility for this algorithm, too, but at the cost of a small increase in matching time.

Table 3 compares the sizes of the new and old indexes in practice. It clearly shows how the space advantage of LZ index grows as the text length increases, although for large q the change is very slow.

10 Concluding Remarks

We have presented a new data structure for quickly finding all occurrences of a given q-gram in a text. The new LZ index works as fast as but requires asymptotically less space than previous q-gram indexes. Our preliminary experiments show that the space advantage is realized already in texts of practical size, although much more work needs to be done to determine when it is actually worth using the LZ index instead of traditional q-gram index. For example, the value of q has a large effect on the size of the LZ index, but, unfortunately, there is very little theoretical and experimental results on the acceptable values of q in the different q-gram methods.

The LZ index cannot be used for all q-gram methods. For example, some q-gram methods use *gapped* q-grams, non-continuous subsequences of length q [15, 3]. For example, the gapped 3-grams of the pattern `MOROGORO` include `OOO` and `RGO`. Other methods use *q-samples*: only every hth q-gram of the text needs to be examined [18, 16, 17]. We leave as an open question whether or not the LZ index can be modified to work efficiently for these variations.

References

[1] R. Baeza-Yates: Space-time trade-offs in text retrieval. In: *Proc. First South American Workshop on String Processing* (ed. R. Baeza-Yates and N. Ziviani), Universidade Federal de Minas Gerais, 1993, 15–21.

[2] R. Boyer and S. Moore: A fast string searching algorithm. *Communications of the ACM* **20**, 10 (1977), 762–772.

[3] A. Califano and I. Rigoutsos: FLASH: A fast look-up algorithm for string homology. In: *Proc. First International Conference on Intelligent Systems for Molecular Biology* (ed. L. Hunter, D. Searls, and J. Shavlik), AAAI Press, 1993, 56–64.

[4] W. Chang and E. Lawler: Sublinear approximate string matching and biological applications. *Algorithmica* **12**, 4–5 (1994), 327–344.

[5] W. Chang and T. Marr: Approximate string matching and local similarity. In: *Combinatorial Pattern Matching, Proceedings of 5th Annual Symposium* (ed. M. Crochemore and D. Gusfield), *Lecture Notes in Computer Science* **807**, Springer-Verlag, Berlin, 1994, 259–273.

[6] R. Grossi and F. Luccio: Simple and efficient string matching with k mismatches. *Information Processing Letters* **33** (1989), 113–120.

[7] N. Holsti and E. Sutinen: Approximate string matching using q-gram places. *Proc. Seventh Finnish Symposium on Computer Science* (ed. M. Penttonen), University of Joensuu, 1994, 23–32.

[8] P. Jokinen and E. Ukkonen: Two algorithms for approximate string matching in static texts. In: *Proceedings of Mathematical Foundations of Computer Science 1991* (ed. A. Tarlecki), *Lecture Notes in Computer Science* **520**, Springer-Verlag, Berlin, 1991, 240–248.

[9] J. Kärkkäinen and E. Ukkonen: Lempel-Ziv parsing and sublinear-size index structures for string matching. In: *Proc. 3rd South American Workshop on String Processing WSP '96.* (eds. N. Ziviani et al.), Carleton University Press, 1996, 141–155.

[10] G. Landau and U. Vishkin: Fast string matching with k differences. *Journal of Computer and System Sciences* **37** (1988), 63–78.

[11] O. Lehtinen, E. Sutinen and J. Tarhio: Experiments on block indexing. In: *Proc. 3rd South American Workshop on String Processing WSP '96.* (eds. N. Ziviani et al.), Carleton University Press, 1996, 183–193.

[12] A. Lempel and J. Ziv: On the complexity of finite sequences. *IEEE Trans. Inform. Theory,* **IT-22**, 1 (Jan. 1977), 75–81.

[13] E. M. McCreight: A space-economical suffix tree construction algorithm. *J. Assoc. Comput. Mach.* **23** (1976), 262–272.

[14] E. Myers: A sublinear algorithm for approximate keyword searching. *Algorithmica* **12**, 4–5 (1994), 345–374.

[15] P. Pevzner and M. Waterman: Multiple filtration and approximate pattern matching. *Algorithmica* **13** (1995), 135–154.

[16] E. Sutinen and J. Tarhio: On using q-gram locations in approximate string matching. In: *Proc. 3rd Annual European Symposium on Algorithms ESA '95* (ed. P. Spirakis), *Lecture Notes in Computer Science* **979**, Springer, Berlin, 1995, 327–340.

[17] E. Sutinen and J. Tarhio: Filtration with q-samples in approximate string matching. In: *Proc. 7th Symposium on Combinatorial Pattern Matching CPM '96* (ed. D. Hirschberg and G. Myers), *Lecture Notes in Computer Science* **1075**, Springer, Berlin, 1996, 50–63.

[18] T. Takaoka: Approximate pattern matching with samples. *Proceedings of ISAAC '94, Lecture Notes in Computer Science* **834**, Springer-Verlag, Berlin, 1994, 234–242.

[19] J. Tarhio and E. Ukkonen: Approximate Boyer-Moore string matching. *SIAM Journal on Computing* **22**, 2 (1993), 243–260.

[20] E. Ukkonen: Finding approximate patterns in strings. *Journal of Algorithms* **6** (1985), 132–137.

[21] E. Ukkonen: Approximate string matching with q-grams and maximal matches. *Theoretical Computer Science* **92**, 1 (1992), 191–211.

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

[23] S. Wu and U. Manber: Fast text searching allowing errors. *Communications of the ACM* **35**, 10 (1992), 83–91.

[24] J. Ziv and A. Lempel: A universal algorithm for sequential data compression. *IEEE Trans. Inform. Theory,* **IT-23**, 3 (May 1977), 337–343.