Lempel-Ziv Parsing and Sublinear-Size Index Structures for String Matching
(Extended Abstract)

Juha Kärkkäinen   Esko Ukkonen

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

Abstract. String matching over a long text can be significantly speeded up with an index structure formed by preprocessing the text. For very long texts, the size of such an index can be a problem. This paper presents the first sublinear-size index structure. The new structure is based on Lempel-Ziv parsing of the text and has size linear in N, the size of the Lempel-Ziv parse. For a text of length n, $N = O(n/\log n)$ and can be still smaller if the text is compressible. With the new index structure, all occurrences of a pattern string of length m can be found in time $O(m^2 + (m+L)\log N + \sqrt{Nm\log m})$, where L is the number of occurrences found.

1 Introduction

The classical *string matching problem* is that of detecting if *pattern* $\mathcal{P}$ occurs in *text* $\mathcal{T}$ (the *existence problem*) and reporting all the occurrences of $\mathcal{P}$ in $\mathcal{T}$ (the *all-occurrences problem*). Here $\mathcal{P}$ and $\mathcal{T}$ are strings in some fixed alphabet Σ.

In this paper, we concentrate on the version of the string matching problem in which preprocessing of text $\mathcal{T}$ is allowed. We assume that $\mathcal{T}$ remains fixed for several searches with varying patterns $\mathcal{P}$. Then, in order to speed up the searches, one can first preprocess $\mathcal{T}$ to build a suitable data structure ("index") that makes the subsequent searches with different $\mathcal{P}$'s considerably faster than would be possible without preprocessing. This is a very central problem of text processing systems and occurs in applications in different areas such as text databases, natural language processing and computational biology.

The well-known data structure *suffix tree* already offers a solution to our problem. A suffix tree for text $\mathcal{T}$ is a trie-like data structure representing all suffixes of $\mathcal{T}$. It can be constructed in time $O(n\log c)$ and space $O(n)$, where $n = |\mathcal{T}|$ is the length of $\mathcal{T}$ and $c = |\Sigma|$ is the size of the alphabet [25, 21, 24]. The existence of an occurrence of $\mathcal{P}$ in $\mathcal{T}$ can be detected using the suffix tree in time $O(m\log c)$ and all occurrences can be listed in time $O(m\log c + L)$, where $m = |\mathcal{P}|$ and L is the number of occurrences.

Although linear in size, a suffix tree can be uneconomically large to be really attractive in practical applications. The size depends on implementation details and the structure of the text, but will never be as low as $10n$ bytes. Suffix arrays

[20, 11] (size $5n$ bytes), level-compressed tries [3, 4] (size about $11n$ bytes), suffix cactuses [15] (size $9n$ bytes), and suffix binary search trees [13] (size about $10n$ bytes) are alternative smaller data structures with almost the same properties as the suffix tree. Their space requirement is still high for large n and it can be impossible to store the entire data structure in the fast memory. Using suffix trees and arrays in secondary memory environment is considered e.g. in [11, 5]. As the slow secondary memory operations can, in practice, destroy the good theoretical performance, there is a need to find small alternatives for suffix trees and arrays, even at the cost of increased search times.

In this paper, we develop the first sublinear-size index structure for text $\mathcal{T}$, based on Lempel-Ziv parsing of $\mathcal{T}$. The size of the structure is $O(N)$, where N is the number of blocks into which LZ parsing divides $\mathcal{T}$. We show that $N < n/(\log_c n - \log_c \log_c n - 1/(c-1)) = O(n/\log n)$. Hence, the size of our index structure is sublinear in the length of $\mathcal{T}$ and is therefore asymptotically smaller than the size of any of the structures mentioned above. A practical implementation takes $18\frac{1}{2}N$ bytes (in addition to $\mathcal{T}$ itself). We expect that, if n is at least about 10^3, our data structure is smaller than the suffix array, which is the smallest of the structures known earlier.

The associated search algorithm for finding the occurrences of $\mathcal{P}$ in $\mathcal{T}$ solves the existence problem in time $O(m^2+m\log N+\sqrt{Nm\log m})$, where $m=|\mathcal{P}|$, and the all-occurences problem in time $O(m^2+(m+L)\log N+\sqrt{Nm\log m})$, where L is the total number of occurrences. The preprocessing time to construct the index structure is $O(n\log c)$. As essential building blocks of the index structure we introduce two new data structures, the 2-d tree for strings and the two-dimensional heap. Both may have applications beyond the present context.

Our search algorithm is based on a special property of LZ parsing of $\mathcal{T}$, already noted in [9]: The first occurrence (from left to right) of $\mathcal{P}$ in $\mathcal{T}$ must overlap the last symbol of some block produced by the LZ parsing. Therefore we divide the occurrences into two types: The *primary occurrences* overlap the last symbol of some LZ block; the remaining occurrences are the *secondary occurrences*. The search algorithm consists of two phases. The *primary search* phase finds the primary occurrences, and the *secondary search* uses the primary occurrences as input and finds the secondary occurrences using the knowledge of the repetitive structure of $\mathcal{T}$ uncovered by LZ parsing. Note that for solving the existence problem the primary search alone is enough.

A totally different possibility to construct a small index structure is to build a *sparse suffix tree*, i.e., a suffix tree that represents only a subset of all suffixes [11, 23, 17, 2]. In this way, all occurrences of $\mathcal{P}$ that overlap the starting point of a suffix that is present in the tree can easily be found. The other occurrences are difficult and lead to a brute-force search. This is in contrast with the method of the present paper that also finds such secondary occurrences fast using the properties of LZ parsing.

The basic two-phase technique based on Lempel-Ziv parsing has also been applied to searching q-grams [16]. The actual data structures, however, are very different from the ones described here. Technically related is also the problem of *compressed matching* [1, 9] which asks one to find the occurrences of $\mathcal{P}$ in $\mathcal{T}$ directly from the LZ compressed representation of $\mathcal{T}$. The problem is quite different but the solutions utilize the same properties of LZ parsing as we do. Finally, we note that the subproblem of finding block matches in the dynamic text indexing algorithm of Gu, Farach and Beigel [12] resembles an on-line (no preprocessing) version of our primary searching.

2 LZ Parsing

The Lempel-Ziv scheme of data compression is based on parsing the string into consecutive disjoint blocks that follow the internal repetitive structure of the string. The scheme occurs in several variations. Our variation is the original from the seminal paper by Lempel and Ziv [19] which preceded the papers describing their famous data compression methods [26, 27]. The notation is borrowed from [9].

We define that the *LZ parse* of string $\mathcal{T}$ is a sequence of the form

$$\mathcal{Z} = (P_1, L_1, C_1) \cdots (P_i, L_i, C_i) \cdots (P_N, L_N, C_N),$$

where each P_i and L_i is a non-negative integer and each C_i is a symbol in the alphabet Σ. Sequence $\mathcal{Z}$ represents $\mathcal{T}$ as follows. Let $U_1 = 1$ and $U_i = U_{i-1} + L_{i-1} + 1$ for $i > 1$. First, it is required that $P_i < U_i$, whenever $L_i > 0$. Moreover, for $1 \le i \le N$,

$$\mathcal{T}[U_i \ldots U_i + L_i - 1] = \mathcal{T}[P_i \ldots P_i + L_i - 1] \quad \text{and} \quad \mathcal{T}[U_i + L_i] = C_i.$$

Each substring $\mathcal{T}[U_i \ldots U_i + L_i]$ is called a *block* of $\mathcal{T}$. The parse $\mathcal{Z}$ gives a unique decomposition of $\mathcal{T}$ into blocks as $\mathcal{T} = \mathcal{T}[U_1 \ldots U_1 + L_1]\mathcal{T}[U_2 \ldots U_2 + L_2] \cdots \mathcal{T}[U_N \ldots U_N + L_N]$. We additionally require that $\mathcal{Z}$ has the following *greedy parsing property.* Assume that during the left-to-right construction of $\mathcal{Z}$ we have $(P_1, L_1, C_1) \cdots (P_{i-1}, L_{i-1}, C_{i-1})$ ready and have to find (P_i, L_i, C_i). Then select $P_i < U_i$ such that the common prefix of $\mathcal{T}[P_i \ldots]$ and $\mathcal{T}[U_i \ldots]$ is the longest possible, and let L_i be the length of the longest common prefix and $C_i = \mathcal{T}[U_i + L_i]$.

Example 1. Let $\mathcal{T} = \mathtt{aaaabbaaababaaabb}$. Then the LZ parse of $\mathcal{T}$ is

$$\mathcal{Z} = (0, 0, \mathtt{a})(1, 3, \mathtt{b})(5, 1, \mathtt{a})(3, 3, \mathtt{a})(6, 5, \mathtt{b})$$

and the decomposition into blocks is $\mathtt{a|aaab|ba|aaba|baaabb}$.

The last symbol C_i of the block $\mathcal{T}[U_i \ldots U_i + L_i]$ is called a *boundary symbol.* The remaining prefix $\mathcal{T}[U_i \ldots U_i + L_i - 1]$ of the block, called a *phrase*, is a string that occurs in $\mathcal{T}$ already before the current block. In the LZ compression the phrase is replaced with a reference to the earlier occurrence $\mathcal{T}[P_i \ldots P_i + L_i - 1]$ which is called the *definition* of the phrase.

The LZ parse $\mathcal{Z}$ can be constructed in $O(n \log c)$ time and $O(n)$ space using a suffix tree, where $n = |\mathcal{T}|$ and $c = |\Sigma|$. Note that, when using the greedy parsing, the last block might not be completed. This could violate the block uniqueness property, which states that no two blocks are identical. This property, which is essential for some of our results*, is a direct consequence of the greedy parsing. The last block problem can be removed by adding a non-alphabet symbol to the end of the text. The extra symbol forces the last block to be completed. In the following, we will assume that the parse $\mathcal{Z}$ has a complete last block.

A useful variation of the above definition of LZ parsing restricts the definitions to start at the beginning of an earlier block, that is, $P_i \in \{U_1, \ldots, U_{i-1}\}$. All of our results, including Theorem 2, still hold for this variation. The advantage of this variation is that the parse can be constructed in $O(N)$ space (and $O(n \log c)$ time) using a sparse suffix tree. The data compression method of Fiala and Greene [10] is based on this variation.

The size of our index structure will be proportional to the length N of $\mathcal{Z}$. We next derive a bound for N (cf. [19]).

Lemma 1. *Let $\mathcal{T}$ be a text over alphabet of size c and let $\mathcal{Z}$ be an LZ parse of $\mathcal{T}$. There are at most c^k blocks of length at most k in $\mathcal{Z}$.*

Proof. For each block $\mathcal{T}[U_i \ldots U_i + L_i]$ of length at most k (i.e. $L_i \leq k-1$), define an *extended block* E_i as $\mathcal{T}[U_i \ldots U_i + k - 1]$. Let E_i and E_j, where $i < j$, be two such extended blocks. When the LZ parser started to form a new block beginning at position U_j, the longest match it found had length $L_j < k$. If E_i and E_j were identical, the parser would have found a longer match. Thus, there cannot be two identical extended blocks and the total number of extended blocks is at most c^k, which, therefore, is also an upper bound for the number of blocks of length at most k. □

Theorem 2. *Let $\mathcal{T}$ be a text of length n over alphabet of size c and let $\mathcal{Z}$ be an LZ parse of $\mathcal{T}$ with N blocks. Then $n > N(\log_c N - 1/(c-1))$.*

Proof. Let $k = \lceil \log_c N \rceil$. Let $\mathcal{Z}$ have

- c blocks of size 1
- $c^2 - c$ blocks of size 2

  $\vdots$
- $c^{k-1} - c^{k-2}$ blocks of size $k-1$ and
- $N - c^{k-1}$ blocks of size k.

*To be precise, Lemma 1 requires the slightly stronger property of uniqueness of extended blocks.

This distribution of block sizes gives the shortest possible text length without a violation of Lemma 1. The resulting text length is then

$$\begin{aligned} n &= kN - \sum_{i=1}^{k-1} c^i = kN - \frac{c^k - c}{c-1} = N\left(\log_c N + \alpha - \frac{c^k - c}{N(c-1)}\right) \\ &= N\left(\log_c N + \alpha - \frac{c^\alpha}{c-1} + \frac{c}{N(c-1)}\right) > N\left(\log_c N + \alpha - \frac{c^\alpha}{c-1}\right), \end{aligned}$$

where $\alpha = k - \log_c N < 1$. It can be shown that $\alpha - c^\alpha/(c-1) \geq -1/(c-1)$, when $c > 1$ and $0 \leq \alpha \leq 1$. Therefore, we have

$$n > N\left(\log_c N - \frac{1}{c-1}\right).$$

□

Corollary 3. *Let N be the number of blocks in an LZ parse of a text of length n. Then it holds that*

$$N < \frac{n}{\log_c n - \log_c \log_c n - \frac{1}{c-1}}.$$

Proof. We can obviously assume that $n \geq c \geq 2$. Let x be the number such that

$$n = x\left(\log_c x - \frac{1}{c-1}\right). \tag{1}$$

Then, by Theorem 2, $N < x$. If $x \leq n$, then $\log_c x - 1/(c-1) < \log_c n$. Substituting this into (1) we get $x > n/\log_c n$. If $x > n$, then trivially $x > n/\log_c n$. Substituting $x > n/\log_c n$ for the second occurrence of x in (1), we get

$$x < \frac{n}{\log_c n - \log_c \log_c n - \frac{1}{c-1}}.$$

□

The worst case bound $N = O(n/\log n)$ is enough for our results. The expected value of N has been shown to be about $nh/\log n$, where h is the entropy of the text, for various versions of LZ parsing and various models of randomness (see, e.g., [14] and references therein). Some empirical measurements of N are given in Table 1. Random text represents the worst case, uncompressible text.

Table 1. Size of the LZ parse for various texts of length 300000.

type	c	N
english	77	40964
DNA	4	31633
random	2	16734
random	8	47554
random	64	88748

3 Outline of the Algorithm

The LZ parse $\mathcal{Z}$ of $\mathcal{T}$ has the interesting property that the first occurrence of any pattern $\mathcal{P}$ has to contain a boundary symbol of $\mathcal{Z}$. The proof is immediate by contradiction. As the boundary symbols occur in text locations $U_i + L_i$ for $1 \leq i \leq N$, we have the following.

Proposition 4 [9]. *If u is the smallest integer such that $\mathcal{P} = \mathcal{T}[u \ldots u+|\mathcal{P}|-1]$, then $U_i + L_i \in [u, u + |\mathcal{P}| - 1]$ for some $1 \leq i \leq N$.* □

An occurrence of $\mathcal{P}$ in $\mathcal{T}$ is a *primary occurrence* if it contains a boundary symbol of parse $\mathcal{Z}$. Otherwise the occurrence is a *secondary occurrence.*

Example 2. The pattern $\mathcal{P}$ = `ab` has primary occurrences in the text $\mathcal{T}$ of Example 1 starting at positions 4 (overlapping the second boundary symbol) and 11 (overlapping the fourth boundary symbol) and secondary occurrences at positions 9 and 15.

It turns out that these two types of occurrences can be found using different techniques. We end up with a two-phase procedure. First, a *primary search* is performed to find all primary occurrences. This search is restricted to the surroundings of the boundary symbols. In Section 4 we will present a data structure to solve the primary search problem. Note that, by Proposition 4, the primary search alone solves the existence problem.

Having found the primary occurrences, the *secondary search* then finds the secondary occurrences. This search relies heavily on the repetitive structure of $\mathcal{T}$ that is uncovered in LZ parsing. In Section 5, we will give two alternative data structures to solve the secondary search problem. One is an old data structure from computational geometry and the other is a new data structure. Both of them have the same properties in secondary searching.

The two phases of the algorithm are totally independent. Either one of them could be changed without affecting the other. Even their data structures are separate to the extent that the text itself is needed only in the primary search.

4 Primary Search

The objective of the primary search is to find all primary occurrences of pattern $\mathcal{P}$ of length m in text $\mathcal{T}$ with LZ parse $\mathcal{Z}$. A primary occurrence of $\mathcal{P}$ contains a boundary symbol C_i at position $U_i + L_i$ for some i. The occurrence is fully specified by the pair (i, u), called a *reference pair*, where $i \in \{1, \ldots, N\}$ is the index of the boundary symbol C_i and $u \in \{1, \ldots, m\}$ is the position of C_i within the occurrence. If the occurrence stretches over more than one boundary symbol, its reference pair is not unique. We define the *canonical reference pair* of a primary occurrence to be the reference pair with the smallest value of i (also the smallest value of u), i.e., the one corresponding to the first boundary symbol within the occurrence.

We will first look at how to do the primary search for a fixed value of u. The problem is then to find the values of i such that (i, u) is a canonical reference pair of a primary occurrence of $\mathcal{P}$. We will forget the 'canonical' part for a moment and just try to find any reference pairs. We will later show how to avoid the non-canonical reference pairs. The problem at hand can now be stated as follows.

Find all $i \in \{1, \ldots, N\}$ such that $\mathcal{T}[U_i{+}L_i{-}u{+}1 \ldots U_i{+}L_i{-}u{+}m] = \mathcal{P}$.

More precisely, we want to have a data structure, formed by preprocessing text $\mathcal{T}$, that can be used for answering the above problem for any pattern $\mathcal{P}$ and any value of $u \in \{1, \ldots, |\mathcal{P}|\}$.

4.1 Two-Dimensional Prefix Matching

The following is an alternative statement for the problem above.

Find all $i \in \{1, \ldots, N\}$ such that

1. $\mathcal{T}[U_i + L_i - u + 1 \ldots U_i + L_i] = \mathcal{P}[1 \ldots u]$ and
2. $\mathcal{T}[U_i + L_i \ldots U_i + L_i + m - u] = \mathcal{P}[u \ldots m]$.

Note that the position u is included in both parts of the pattern, assuring their non-emptiness. With only the second condition, a sparse suffix tree restricted to suffixes $\mathcal{T}[U_i + L_i + 1 \ldots]$, $i = 1, \ldots, N$, would be a good data structure for the problem. The problem with only the first condition could be solved similarly. With both conditions the problem is essentially two-dimensional and can be stated in a more abstract form as follows.

Problem 1 Two-dimensional prefix matching (2DPM). Given a set of string pairs $\{(\mathcal{X}_i, \mathcal{Y}_i)\}_{i=1,\ldots,N}$ and a query string pair $(\mathcal{X}, \mathcal{Y})$, find all pairs $(\mathcal{X}_i, \mathcal{Y}_i)$ such that $\mathcal{X}$ is a prefix of $\mathcal{X}_i$ and $\mathcal{Y}$ is a prefix of $\mathcal{Y}_i$.

In the case of primary search, we have for $i = 1, \ldots, N$

$$\begin{aligned} \mathcal{X}_i &= \mathcal{T}[U_i \ldots U_i + L_i]^R = \mathcal{T}[U_i \ldots U_{i+1} - 1]^R \\ \mathcal{Y}_i &= \mathcal{T}[U_i + L_i \ldots] = \mathcal{T}[U_{i+1} - 1 \ldots], \end{aligned}$$

where $\mathcal{S}^R$ means the reverse of $\mathcal{S}$ and U_{N+1} is defined as $n + 1$. Similarly, we define $\mathcal{X} = \mathcal{P}[1 \ldots u]^R$ and $\mathcal{Y} = \mathcal{P}[u \ldots m]$. Note that $\mathcal{X}_i$ is defined as the reverse of a *block* while $\mathcal{Y}_i$ is defined as a whole suffix. This guarantees that only the canonical reference pairs will be found.

Example 3. For the text of Example 1, the string pairs $(\mathcal{X}_i, \mathcal{Y}_i)$ are $(\mathtt{a}, \mathtt{aaaabb}\ldots)$, $(\mathtt{baaa}, \mathtt{bbaaab}\ldots)$, $(\mathtt{ab}, \mathtt{aaabab}\ldots)$, $(\mathtt{abaa}, \mathtt{abaaabb})$ and $(\mathtt{bbaaab}, \mathtt{b})$.

We have now reduced the primary search to solving the 2DPM problem for m different query pairs (the different values of u). The 2DPM problem is very similar to the two-dimensional range query which asks for points (x_i, y_i) with x_i in range $[x_l, x_r]$ and y_i in range $[y_l, y_r]$. In the 2DPM, a query string $\mathcal{X}$ can be interpreted as the range $[\mathcal{X}, \mathcal{X}zzz\ldots]$ in the lexicographic ordering, where z is the last character in the alphabet. We will next describe a well-known data structure for solving the two-dimensional range query and then modify it for 2DPM.

4.2 2-D Tree for Strings

The *2-d tree* is a binary search tree for points in two dimensions. It is the two-dimensional case of the multidimensional binary search tree or k-d tree of Bentley [6]. The 2-d tree is like ordinary binary search tree except the odd levels use the x-coordinate as the discriminator and the even levels use the y-coordinate as the discriminator. The 2-d tree for N points can be constructed in $O(N \log N)$ time [6]. The range query in a 2-d tree works in the obvious way by continuing the search in one or both subtrees of each node depending on the result of comparisons in the node. Lee and Wong [18] have shown that the range query in a balanced 2-d tree of N points takes $O(\sqrt{N}+l)$ time in the worst case, where l is the size of the answer.

As noted earlier, the 2DPM problem can be interpreted as a two-dimensional range query. The 2-d tree only assumes the existence of a total ordering on the coordinates and can therefore be applied for the 2DPM problem by using lexicographic ordering. However, due to $O(m)$ time string comparisons the search time increases to $O(m\sqrt{N}+l)$. Fortunately, this can be improved on by using a technique from one-dimensional prefix matching.

The searching in a suffix array [20, 11] is essentially binary searching. With naive string comparisons the search time in a suffix array of N strings would be $O(m \log N + l)$, where l is the size of the answer. With precomputed information about the longest common prefixes (LCPs) between some of the strings, Manber and Myers [20] have reduced the time to $O(m + \log N + l)$. The technique has also been described for the general suffix binary search tree [13]. The LCP technique also applies for the 2-d tree which is a two-dimensional binary tree. The 2DPM search time reduces to $O(m + \sqrt{N} + l)$.

We will not present the technique in more detail in this extended abstract but we will give a short description of the resulting data structure. The pairs are stored in the leaves of a balanced binary tree. The internal nodes contain the discriminating values for $\mathcal{X}$- and $\mathcal{Y}$-coordinates in alternating levels as in a normal 2-d tree. We use the largest value in the left subtree as the discriminating value. Additionally, every node contains the necessary LCP information. A practical implementation of the data structure requires $8\frac{1}{2}N$ bytes. Without the LCP information, the space requirement could be reduced to $4N$ bytes.

4.3 Construction

The 2-d tree can be constructed in $O(N \log N)$ time. As with searching, a naive application of the construction for a 2-d tree for strings would increase the time by a factor m. This can be improved on as follows. We first construct the suffix arrays for the two coordinates separately. Let x_i and y_i denote the positions of $\mathcal{X}_i$ and Y_i in their corresponding suffix array. Then it holds that $x_i < x_j$ if and only if X_i is lexicographically smaller than X_j. We can therefore construct a 2-d tree for the position pairs (x_i, y_i) using Bentley's $O(N \log N)$-time algorithm and then replace the position pairs with the strings. Below, we will describe how to construct the two suffix arrays in $O(n \log c)$ time. The rest of the construction takes time $O(N \log N)$ which is $O(n)$ by Theorem 2. Therefore the total construction time is $O(n \log c)$.

It remains to show how to construct the suffix arrays in $O(n \log c)$ time. The normal suffix array construction algorithms do not work here, because these suffix arrays do not have all suffixes (or reverse prefixes) of the text. However, based on the following lemma, we will show that a brute force construction is fast enough in this case.

Lemma 5. *Let $lcp(i,j)$ be the length of the LCP (longest common prefix) of suffixes $\mathcal{T}[U_i + L_i \ldots]$ and $\mathcal{T}[U_j + L_j \ldots]$. Then*

$$\sum_{i=1}^{N} \max_{j \neq i} lcp(i,j) \leq 2n.$$

Proof. Due to the greedy parsing property, $lcp(i,j) \leq L_j + 1$ for $1 \leq i < j \leq N$. For each i, let j_i be such that $lcp(i, j_i)$ is maximized. If $lcp(i, j_i) \leq L_i + 1$, we charge i for it. Otherwise, $lcp(i, j_i) \leq L_{j_i} + 1$ and we charge j_i for it. We will show that each i is charged at most twice. This proves the result because $\sum_{i=1}^{N}(L_i + 1) = n$.

Let i and i' be such that $j_i = j_{i'} = j$. Then $lcp(i, i') \leq \max\{L_i, L_{i'}\} + 1$ and thus, $lcp(i,j) > L_i + 1$ and $lcp(i', j) > L_{i'} + 1$ cannot both be true. Therefore, for each j, there is at most one $i \neq j$ such that j is charged for $lcp(i, j_i)$. □

The suffixes in the lemma are the $\mathcal{Y}$-coordinates of the 2DPM problem. The same result holds trivially for the $\mathcal{X}$-coordinates, which are reverse blocks of the text. As a consequence, doing a constant number of LCP comparisons for every string requires $O(n)$ time.

Our suffix array construction has the following steps.

1. Build a suffix tree for the strings using the brute force method of adding the strings to the tree one at a time starting from the root. The number of steps traveled in the tree when adding string $\mathcal{X}$ is the maximum length of the LCP of $\mathcal{X}$ and some string already in the tree. Each step in the tree takes $O(\log c)$ time. By Lemma 5, the total time is then $O(n \log c)$.

2. Compute the lexicographical ordering of the strings from the suffix tree by a lexicographical depth-first traversal in $O(N)$ time.

3. Compute the LCP-values by string comparisons. For a detailed description of the LCP-values we refer to [20]. For the present purpose, it is enough to know that to compute the LCP-values for position i, we need to find the

LCPs between the string at position i and two other strings. By Lemma 5, the total time taken by this step is then $O(n)$.

The whole construction can be done in $O(N)$ space in addition to text $\mathcal{T}$.

4.4 Improving Total Primary Search Time

The primary search consist of m 2DPM queries. Using the above 2DPM algorithm gives search time $O(m^2 + m\sqrt{N} + L_1)$, where L_1 is the total number of primary occurrences. The term $m\sqrt{N}$ can be improved on by suitably combining some of the queries.

Let us first consider the general range query. A query $([x_l, x_r], [y_l, y_r])$ specifies a rectangular region. Two regions are *one-dimensionally overlapping (1-overlapping)* if either the x-ranges or the y-ranges overlap. Doing m range searches in a 2-d tree of size N for m non-1-overlapping regions needs time $O(\sum_{i=1}^{m} \sqrt{N_i})$ (plus the size of the result), where $\sum_{i=1}^{m} N_i \leq N$. In the worst case the time is $O(\sqrt{mN})$. An *overlap set* is a set of pairwise 1-overlapping query regions. One region may belong to more than one overlap set. Let k_i, $1 \leq i \leq m$, be the size of the maximum overlap set containing the ith query region. Then, the worst case search time can be shown to be $O(\sum_{i=1}^{m} \sqrt{k_i N_i})$. If there is an overlap set of size $\Theta(m)$, the search time can be $\Theta(m\sqrt{N})$.

Two query string pairs $(\mathcal{X}, \mathcal{Y})$ and $(\mathcal{X}', \mathcal{Y}')$ are 1-overlapping if either, one of $\mathcal{X}$ and $\mathcal{X}'$, or one of $\mathcal{Y}$ and $\mathcal{Y}'$ is a prefix of the other. The worst case with respect to 1-overlaps rises when either the pattern, or a long prefix or suffix of the pattern is highly periodic. However, the search for the query pairs in an overlap set can be combined if the overlaps are due to periodicity. We call such an overlap set *periodic*. The combined search takes advantage of the fact that the periodicity causes similarities in both coordinates. In general, a 1-overlap in one coordinate does not imply similarity in the other coordinate. However, it can be shown that the largest possible non-periodic overlap set is of size $O(\log m)$. As a result the total primary search time can be reduced to $O(m^2 + m \log N + \sqrt{Nm \log m} + L_1)^{\star\star}$.

To find the interesting periodicities in the pattern, we use a data structure called the *border tree* [12]. The border tree was originally used for answering prefix–suffix queries, which is indeed closely related to finding periodicities. We build two border trees, one for the prefixes and the other for the suffixes of the pattern. Each node of the suffix (prefix) border tree represents one or more suffixes (prefixes). The nodes representing more than one suffix (prefix) correspond to a periodic overlap set. A non-periodic overlap set corresponds to a path in one of the border trees. The $O(\log m)$ maximum for the largest non-periodic overlap set is a direct consequence of the fact that the height of the border tree is $O(\log m)$. The border tree can be constructed and the periodic overlap sets formed in time $O(m)$.

**The $m \log N$ term was earlier hidden by the $m\sqrt{N}$ term.

We will not give further details of the above method or proofs of the results in this extended abstract. As a final note, we mention that the term $\sqrt{Nm \log m}$ is realized only in a rare special case. The actual primary search time depends in a complicated way on both the pattern and the text. We believe that the average case search time is significantly below the worst case given here.

5 Secondary Search

The objective of the secondary search is to find all secondary occurrences of a known occurrence of pattern $\mathcal{P}$ in text $\mathcal{T}$ with LZ parse $\mathcal{Z}$. A known occurrence $\mathcal{O} = \mathcal{T}[u \ldots u + m - 1]$ of $\mathcal{P}$ has a secondary occurrence within a phrase $\mathcal{T}[U_i \ldots U_i + L_i - 1]$ if the corresponding definition $\mathcal{T}[P_i \ldots P_i + L_i - 1]$ contains $\mathcal{O}$. The problem can be stated as follows.

> Find all $i \in \{1, \ldots, N\}$ such that the definition $\mathcal{T}[P_i \ldots P_i + L_i - 1]$ contains the occurrence $\mathcal{O} = \mathcal{T}[u \ldots u + m - 1]$, i.e. $P_i \leq u$ and $P_i + L_i - 1 \geq u + m - 1$.

For each such i, there is a secondary occurrence $\mathcal{T}[U_i - P_i + u \ldots U_i - P_i + u + m - 1]$.

The secondary search will be made for all occurrences found, including all the secondary occurrences. We want to have a data structure, formed by preprocessing the LZ parse $\mathcal{Z}$, that can answer the problem efficiently for any occurrence.

Note that we do not need access to the text itself during secondary search. We already know the relevant facts that $\mathcal{T}[u \ldots u + m - 1] = \mathcal{P}$ and $\mathcal{T}[P_i \ldots P_i + L_i - 1] = \mathcal{T}[U_i \ldots U_i + L_i - 1]$ for all i. This observation gives us the following abstract formulation of the problem.

Problem 2 Interval containment. Given a set of N intervals $\{[x_i, y_i]\}_{i=1,\ldots,N}$, and a query interval $[x, y]$, find all $[x_i, y_i]$ that contain $[x, y]$, i.e., $x_i \leq x$ and $y_i \geq y$.

In the case of secondary search, we have $x_i = P_i$, $y_i = P_i + L_i - 1$, $x = u$ and $y = u + m - 1$. To locate the secondary occurrences, each interval $[x_i, y_i]$ has an associated offset value $o_i = U_i - P_i$. If $[x_i, y_i]$ contains $[x, y]$, then $\mathcal{T}[x + o_i \ldots y + o_i]$ is a secondary occurrence.

Example 4. For the text $\mathcal{T}$ of Example 1, the intervals $[x_i, y_i]$ are $[1, 3]$, $[5, 5]$, $[3, 5]$ and $[6, 10]$, and the corresponding offsets o_i are 1, 1, 5 and 6. The first phrase of $\mathcal{T}$ is empty and the corresponding interval is omitted here. In fact, the first phrase is always empty.

5.1 Priority Search Tree

As the last inequations in the problem statement above show, the 2DPM problem can also be seen as a special case of a two-dimensional semi-infinite range query. In the general case, one does not assume that the pairs are intervals ($x \leq y$). Another closely related computational geometry problem is the interval intersection problem, for which there exists a number of data structures such as the segment tree [7], the interval tree [8] and the priority search tree [22]. The last of the three is the most suitable for our purposes due to its small space requirement.

The *priority search tree*, invented by McCreight [22], was originally designed for solving semi-infinite range queries in two-dimensional space. With the priority search tree the interval containment problem with N intervals can be answered in time $O(\log N + l)$, where l is the size of the result. The size of the priority search tree is $O(N)$ and it can be constructed in $O(N \log N)$ time. Furthermore, the priority search tree can be implemented so that it requires no space in addition to the intervals and their associated offsets.

5.2 2-D Heap

The 2-d heap is a new data structure that can answer the interval containment problem in time $O(\log N + l \log N)$. It is thus slower than the priority search tree, when the size of the answer is large. However, the secondary search must be done for every occurrence found. Therefore, for both the priority search tree and the 2-d heap, the total execution time of the secondary searches over the whole algorithm is $O(L \log N)$, where L is the total number of occurrences of the pattern. Like the priority search tree, the 2-d heap can be implemented in no space in addition to the intervals and their associated offsets. The simplicity of the 2-d heap may make it a better alternative in practice.

The 2-d heap is a simple binary tree structure, where each node of the tree contains one of the intervals. The interval $[x_i, y_i]$ in a node has the property that all intervals in the left subtree of the node have smaller or equal y-values and all intervals in the right subtree have larger or equal x-values. The y-value can be seen as the top of the heap for the left subtree and the x-value the top of the heap for the right subtree.

The search for the interval containment problem goes as follows. When the query interval $[x, y]$ is compared to the root interval (of a subtree), the following cases are possible.

1. $x_i \le x$ and $y_i \ge y$. The root interval contains the query interval. The search continues in both subtrees.

2. $x_i \le x$ and $y_i < y$. The root interval is too far to the left. The left subtree can now be eliminated because all intervals in it are even more to the left. The search continues only in the right subtree.

3. $x_i > x$ and $y_i \ge y$. The root interval is too far to the right. The right subtree can now be eliminated because all intervals in it are even more to the right. The search continues only in the left subtree.

4. $x_i > x$ and $y_i < y$. The root interval is contained by the query interval. Both subtrees can be eliminated because all intervals in the left subtree have too small y-value and all intervals in the right subtree have too large x-value.

The answer to the interval containment problem is the set of intervals in the nodes of type 1. The following lemma shows then that the algorithm solves the interval containment problem in time $O((1+l)\log N)$, where l is the size of the answer.

Lemma 6. *The above algorithm accesses at most $1+(1+l)\log N$ nodes in a balanced 2-d heap of N nodes and l nodes of type 1.*

Proof. We prove the result by induction on N. Let $T(N,l)$ be the number of nodes accessed. The result is trivially true, if $N=1$ or the root is of type 4. If the root is of type 2 or 3, the search continues in one subtree only and the following holds.

$$T(N,l) \leq 1+T(N/2,l) \leq 2+(1+l)\log(N/2) \leq 1+(1+l)\log N.$$

In the case 1, the search continues in both subtrees. Let l_l and l_r be the number of nodes of type 1 in the two subtrees. Then $l=1+l_l+l_r$ and

$$\begin{aligned} T(N,l) &\leq 1+T(N/2,l_l)+T(N/2,l_r) \leq 3+(2+l_l+l_r)\log(N/2) \\ &\leq 1+(2+l_l+l_r)\log N \leq 1+(1+l)\log N. \end{aligned}$$

□

An array representation of a perfectly balanced 2-d heap can be constructed in time $O(N\log N)$ as follows. The intervals are sorted by the x-values. The first half is then searched for the largest y-value and the corresponding interval is moved to the middle position (the root). All intervals between the old and the new position are shifted one step to the left. The root interval now satisfies the condition that the intervals to the left of the root have smaller or equal y-values and the intervals to the right of the root have larger or equal x-values. The selection of the root is repeated recursively for each subtree.

6 Summary

We have presented a new method for string matching with preprocessing of the text. The method uses LZ parsing to divide the string matching problem into two subproblems, the primary and the secondary searches. We gave separate solutions for the subproblems. The key properties of the resulting combined data structure are summarized in Table 2.

There are several possibilities for variations in our basic approach. A different version of LZ parsing can be used; we already suggested one in Section 2. The solutions to the primary and secondary searches can be changed, independent of each other. This may allow, for example, methods with different space-time trade-offs. A very different variation, in all aspects, applied to a slightly different problem is given in [16].

Table 2. The key properties of the whole index. Here n is the length of the text, m the length of the pattern, N the number of blocks in the LZ parse, c the size of the alphabet, and L the total number of occurrences.

size (bytes)	$18\frac{1}{2}N$
construction time	$O(n \log c)$
search time for the existence problem	$O\left(m^2 + m \log N + \sqrt{Nm \log m}\right)$
search time for the all-occurrences problem	$O\left(m^2 + (m+L) \log N + \sqrt{Nm \log m}\right)$

References

1. A. Amir, G. Benson, and M. Farach. Let sleeping files lie: Pattern matching in Z-compressed files. In *Proc. 5th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 705–714, 1994.

2. A. Andersson, N. J. Larsson, and K. Swansson. Suffix trees on words. In *Proc. 7th Symposium on Combinatorial Pattern Matching (CPM)*, 1996. To appear.

3. A. Andersson and S. Nilsson. Improved behaviour of tries by adaptive branching. *Inf. Process. Lett.*, 46(6):295–300, July 1993.

4. A. Andersson and S. Nilsson. Efficient implementation of suffix trees. *Software—Practice and Experience*, 25(2):129–141, Feb. 1995.

5. E. F. Barbosa, G. Navarro, R. Baeza-Yates, C. Perleberg, and N. Ziviani. Optimized binary search and text retrieval. In *Proc. 3rd Annual European Symposium on Algorithms, ESA '95*, pages 311–326, 1995.

6. J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, Sept. 1975.

7. J. L. Bentley and D. Wood. An optimal worst-case algorithm for reporting intersections of rectangles. *IEEE Trans. Computers*, 29(7):571–577, July 1980.

8. H. Edelsbrunner. A new approach to rectangle intersections. *International Journal of Computer Mathematics*, 13(3–4):209–229, 1983.

9. M. Farach and M. Thorup. String matching in Lempel-Ziv compressed strings. In *Proc. ACM Symposium on Theory of Computing (STOC)*, pages 703–712, 1995.

10. E. R. Fiala and D. H. Greene. Data compression with finite windows. *Commun. ACM*, 32(4):490–505, Apr. 1989.

11. G. H. Gonnet, R. A. Baeza-Yates, and T. Snider. Lexicographical indices for text: Inverted files vs. PAT trees. Technical Report OED-91-01, Centre for the New OED, University of Waterloo, 1991.

12. M. Gu, M. Farach, and R. Beigel. An efficient algorithm for dynamic text indexing. In *Proc. 5th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 697–704, 1994.

13. R. W. Irving. Suffix binary search trees. Technical report TR-1995-7, Computing Science Department, University of Glasgow, Apr. 1995.

14. P. Jacquet and W. Szpankowski. Asymptotic behavior of the lempel-ziv parsing scheme and digital search trees. *Theoretical Comput. Sci.*, 144(1–2):161–197, June 1995.

15. J. Kärkkäinen. Suffix cactus: A cross between suffix tree and suffix array. In *Proc. 6th Symposium on Combinatorial Pattern Matching, CPM 95*, pages 191–204, 1995.

16. J. Kärkkäinen and E. Sutinen. Lempel-Ziv index for q-grams. Manuscript, 1996.

17. J. Kärkkäinen and E. Ukkonen. Sparse suffix trees. In *Proc. 2nd Annual International Computing and Combinatorics Conference (COCOON)*, 1996. To appear.

18. D. T. Lee and C. K. Wong. Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees. *Acta Inf.*, 9(1):23–29, Jan. 1977.

19. A. Lempel and J. Ziv. On the complexity of finite sequences. *IEEE Trans. Inform. Theory*, IT-22(1):75–81, Jan. 1976.

20. U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, Oct. 1993.

21. E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, Apr. 1976.

22. E. M. McCreight. Priority search trees. *SIAM J. Comput.*, 14(2):257–276, May 1985.

23. D. R. Morrison. PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric. *J. ACM*, 15(4):514–534, Oct. 1968.

24. E. Ukkonen. On-line construction of suffix-trees. *Algorithmica*, 14(3):249–260, Sept. 1995.

25. P. Weiner. Linear pattern matching algorithms. In *Proc. IEEE 14th Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.

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

27. J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. Inform. Theory*, IT-24(5):530–536, Sept. 1978.

This article was processed using the LaTeX 2_ε macro package with CUP_CS class