Crochemore's String Matching Algorithm: Simplification, Extensions, Applications*

Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi

Department of Computer Science,
University of Helsinki
Helsinki, Finland
{*firstname.lastname*}@cs.helsinki.fi

Abstract. We address the problem of string matching in the special case where the pattern is very long. First, constant extra space algorithms are desirable with long patterns, and we describe a simplified version of Crochemore's algorithm retaining its linear time complexity and constant extra space usage. Second, long patterns are unlikely to occur in the text at all. Thus we define a generalization of string matching called Longest Prefix Matching that asks for the occurrences of the longest prefix of the pattern occurring in the text at least once, and modify the simplified Crochemore's algorithm to solve this problem. Finally, we define and solve the problem of Sparse Longest Prefix Matching that is useful when the pattern has to be split into multiple pieces because it is too long to be processed in one piece. These problems are motivated by and have application in Lempel-Ziv (LZ77) factorization.

1 Introduction

String matching, the problem of finding all the occurrences of a string $\mathsf{Y}[0..m)$ (the pattern) in a larger string $\mathsf{X}[0..n)$ (the text)[1] is a foundational problem in computer science, and has applications throughout modern computer software. Several algorithms that are optimal in space ($O(1)$ extra space) as well as in time ($O(n+m)$) were discovered in the 80's and 90's [11,10,2,3]. In practice, these algorithms, though optimal in theory, are greatly outperformed by algorithms that use $O(m)$ extra space [5], and thus to date have been mostly a theoretical curiousity. Our own interest in constant extra space algorithms is, however, a practical one: the space-efficient computation of the LZ77 factorization of a string [20]. To our knowledge, this is the first practical application to make use of these optimal string matching techniques, and the first time they have been applied to problems beyond simple pattern matching.

The LZ77 factorization of large strings has many important applications these days, for example in compression [6,12] and indexing [9,8,18] of large text collections (see [16] for more applications of LZ77). The factorization can be computed in linear time but at the cost of using a lot of space [8,15]. We have recently introduced a more space-efficient approach running in $O(nd)$ time while using $O(n/d)$ space. The space requirement is in addition to the text when operating in main memory [14], and in total when using external memory [13]. However, very long phrases are a problem for the basic approach and have to be processed differently. There are at most $O(d)$ of such long phrases and thus we can afford to spend $O(n)$ time for each. As mentioned in [14,13], the long phrase computation is based on a modified Crochemore's algorithm

* This research is partially supported by the Academy of Finland thorugh grant 118653 (ALGODAN) and grant 250345 (CoECGR).

[1] We write $[i..j)$ as a shorthand for $[i..j-1]$.

Juha Kärkkäinen, Dominik Kempa, Simon J. Puglisi: Crochemore's String Matching Algorithm: Simplification, Extensions, Applications, pp. 168–175.
Proceedings of PSC 2013, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05330-0

for string matching but it is never described in more detail. In this paper, we formulate the long phrase computation task as two more general formal problems and show how to solve them by modifying Crochemore's algorithm.

The critical operation in computing the factorization is to find the longest prefix of suffix $\mathsf{X}[i..n)$ that occurs at the some earlier position $j < i$ in X. If the length of this prefix is ℓ, then the next factor will be a prefix of $\mathsf{X}[i+\ell..n)$. If we consider suffix $\mathsf{X}[i..n)$ as a pattern, we can formulate the operation as a special case of the following more general problem.

Definition 1. *Given two strings, a text and a pattern, the* Longest Prefix Matching *problem is to find the length of the longest prefix of the pattern that occurs in the text and to report all occurrences of that prefix in the text.*

From now on we will only consider this general problem. However, consistent with the application in LZ factorization, we focus on the case where the pattern and even the matching prefix is extremely long.

Note that if the pattern as a whole occurs in the text, the output is the occurrences of the pattern. Thus Longest Prefix Matching is a generalization of standard exact string matching. String matching algorithms based on matching pattern prefixes such as Knuth–Morris–Pratt (KMP) [17] can be easily modified to perform Longest Prefix Matching, while others such as Boyer–Moore [1] cannot. However, when the pattern is very long, the space requirement of the data structures built during KMP preprocessing can become a problem. Among the constant extra space algorithms that we are aware of, Crochemore's algorithm [2] is the only one based on matching pattern prefixes. Thus it is the basis of our solution to the Longest Prefix Matching problem. Crochemore's algorithm, and particularly its analysis, is quite complicated. Our first contribution is a simplified version of the algorithm that retains the linear time complexity and constant extra space usage. We then generalize the simple version to solve the Longest Prefix Matching problem in the same time and space complexity.

Even Crochemore's algorithm needs fast access to the full pattern, but in the external memory context the pattern length may even exceed the size of the available memory. To deal with this case, we split the pattern into blocks $\mathsf{Y} = \mathsf{Y}[0..M)\mathsf{Y}[M..2M)\ldots$ that are small enough to fit in memory. We start with longest prefix matching for the first block. If the full block occurs in the text, we then process the second block but considering only occurrences that start where an occurrence of the previous block ends. We continue to process further blocks in the same way as long as necessary. The matching problem for the second and further blocks can be formulated as the following general problem:

Definition 2. *Given two strings, a text and a pattern, and an ascending sequence of text positions, the* Sparse Longest Prefix Matching *problem is to find the length of the longest prefix of the pattern that occurs in the text starting at one of the specified positions and to report all such occurrences.*

We generalize Crochemore's algorithm to solve this problem too.

2 Preliminaries

Strings. Consider a string $\mathsf{X} = \mathsf{X}[0..n-1] = \mathsf{X}[0]\mathsf{X}[1]\cdots\mathsf{X}[n-1]$ of $|\mathsf{X}| = n$ symbols drawn from an ordered alphabet Σ of size σ. For $i = 0, \ldots, n-1$ we write X_i to denote the *suffix* of X of length $n-i$, that is $\mathsf{X}_i = \mathsf{X}[i..n-1] = \mathsf{X}[i]\mathsf{X}[i+1]\cdots\mathsf{X}[n-1]$. The lexicographically maximal among all suffixes of X is denoted $\mathsf{MS}(\mathsf{X})$. By $\mathsf{lcp}(\mathsf{X}, \mathsf{Y})$ we denote the length of the longest common prefix of X and Y. A string Y is said to be a *border* of X if Y is both a prefix and a suffix of X. A string is called *border-free* if it has no borders, except itself and the empty string.

Periods. A positive integer p is called a *period* of X if $\mathsf{X}[i] = \mathsf{X}[i+p]$ for any $i \in [0..n-p)$. The shortest period of X is denoted $per(\mathsf{X})$. We say that X is *k-periodic* if $per(\mathsf{X}) \leq |\mathsf{X}|/k$. Throughout we use a classic result about periodicity due to Fine and Wilf [7].

Lemma 1 (Weak Periodicity Lemma) *If a string X has periods p and q that satisfy $p + q \leq |\mathsf{X}|$ then X also has period $\gcd(p, q)$.*

3 Simplified Crochemore's Algorithm

Crochemore's algorithm resembles in many ways the famous Morris-Pratt [19] (MP in short) algorithm[1]. At a generic step it attempts to match the pattern Y against the suffix X_i of the text by computing $\ell = \mathsf{lcp}(\mathsf{X}_i, \mathsf{Y})$ and checking whether $\ell = m$. After that it determines the next position $i + q$ in the text at which the pattern may occur. The value of ℓ is then either set to zero or - if partial information about $\mathsf{lcp}(\mathsf{X}_{i+q}, \mathsf{Y})$ is known - to a positive value in order to speed up the next lcp query. Note that any shift length q satisfying $q \leq per(\mathsf{Y}[0..\ell))$ is safe, i.e., prevents from missing an occurrence of Y due to the following fact.

Observation 2 *Assume $\mathsf{X}_i[0..\ell) = \mathsf{Y}[0..\ell)$. Then for any $k \in [1..per(\mathsf{Y}[0..\ell)))$ it holds $\mathsf{lcp}(\mathsf{X}_{i+k}, \mathsf{Y}) = \mathsf{lcp}(\mathsf{Y}_k, \mathsf{Y}) < \ell - k$.*

The main difference between MP and Crochemore's algorithm is the choice of shift length q and how it is computed. MP precomputes and stores $per(\mathsf{Y}[0..i))$ for all $i \in [1..m]$, and always sets $q = per(\mathsf{Y}[0..\ell))$ (or $q = 1$ if $\ell = 0$). Crochemore's algorithm uses only $O(1)$ extra space in addition to the text and the pattern (which are treated as read-only) thus cannot afford to store these values. Instead, as the computation of $\mathsf{lcp}(\mathsf{X}_i, \mathsf{Y})$ is taking place, it is simultaneously computing the lexicographically maximal suffix (together with its shortest period) of the growing pattern prefix that matches the text.

Fig. 1 shows an algorithm, called `UpdateMS`, that updates the maximal suffix computation when the prefix match is extended by one character. It is based on properties of maximal suffixes observed by Duval [4] and detailed in the following theorem.

[1] We point out that the original Crochemore's algorithm performs slightly more complicated shifts than MP making it closer to KMP [17] algorithm.

Function `UpdateMS`(Y, ℓ, s, p)

Input: a string Y and integers ℓ, s, p such that
$\mathsf{MS}(\mathsf{Y}[0..\ell)) = \mathsf{Y}[s..\ell)$ and $p = per(\mathsf{Y}[s..\ell))$.
Output: a triple $(\ell+1, s, p)$ such that
$\mathsf{MS}(\mathsf{Y}[0..\ell+1)) = \mathsf{Y}[s..\ell+1)$ and $p = per(\mathsf{Y}[s..\ell+1))$.

```
1:  if ℓ = 0 then
2:      return (1, 0, 1)
3:  i ← ℓ
4:  while i < ℓ + 1 do
        // MS(Y[0..i)) = Y[s..i) and p = per(Y[s..i))
        // A = Y[s..s + p) and B = Y[i − (i − s) mod p..i)
5:      if Y[i − p] > Y[i] then // Theorem 3, case (3)
6:          i ← i − (i − s) mod p
7:          s ← i
8:          p ← 1
9:      elsif Y[i − p] < Y[i] then // Theorem 3, case (2)
10:         p ← i − s + 1
11:     i ← i + 1
12: return (ℓ + 1, s, p)
```

Figure 1. A procedure extending the matching pattern prefix by one letter simultaneously updating its maximal suffix and associated shortest period.

Theorem 3 *Let* $\mathsf{Y} = \mathsf{PA}^k\mathsf{B}$ *where* $\mathsf{M} := \mathsf{MS}(\mathsf{Y}) = \mathsf{A}^k\mathsf{B}$ *and* $|\mathsf{B}| < |\mathsf{A}| = p := per(\mathsf{MS}(\mathsf{Y}))$. *Suppose* $a \in \Sigma$ *is such that* $\mathsf{B}a$ *is a prefix of* A *and* b *is an arbitrary character. Then* $\mathsf{M}_b := \mathsf{MS}(\mathsf{Y}b)$ *and* $p_b := per(\mathsf{M}_b)$ *satisfy*

$$\mathsf{M}_b = \mathsf{M}b \text{ and } p_b = p \qquad \text{if } a = b \tag{1}$$
$$\mathsf{M}_b = \mathsf{M}b \text{ and } p_b = |\mathsf{M}b| \qquad \text{if } a < b \tag{2}$$
$$\mathsf{M}_b = \mathsf{MS}(\mathsf{B}b) \qquad \text{if } a > b \tag{3}$$

A key to easily proving this theorem is a simple fact about maximal suffixes:

Lemma 4 *Let* $\mathsf{Y} = \mathsf{PA}^k\mathsf{B}$, *where* $\mathsf{MS}(\mathsf{Y}) = \mathsf{A}^k\mathsf{B}$ *and* $|\mathsf{B}| < |\mathsf{A}| = per(\mathsf{MS}(\mathsf{Y}))$. *The string* A *is border-free.*

Observe that each step of the while loop on line 4 in `UpdateMS` increases the value of the non-decreasing expression $i + s$. The final and initial values of i differ exactly by one. Hence we can make the following observation.

Observation 5 *The cost of* `UpdateMS` *is* $O(\Delta s)$.

The key property of maximal suffixes is the connection between $per(\mathsf{Y}[0..\ell))$ and $per(\mathsf{MS}(\mathsf{Y}[0..\ell)))$. In certain (easy to recognize) situations the two values are equal. We will now give a precise description of this connection.

We point out that a superset of the properties stated next is proven in [2]. However, our version of the algorithm requires a smaller number of (slightly simpler, both in terms of the claim and the proof) formal statements and we leave the proofs to present the algorithm description standalone.

Algorithm `Match`$(\mathsf{X}, n, \mathsf{Y}, m)$
Input: strings $\mathsf{X}[0..n)$ (text) and $\mathsf{Y}[0..m)$ (pattern).
Output: the set $\mathcal{S} = \{i \in [0..n) \mid \mathsf{X}[i..i+m) = \mathsf{Y}\}$.

```
1:  S ← ∅
2:  i ← ℓ ← p ← s ← 0
3:  while i < n do
4:      while i + ℓ < n and ℓ < m and X[i + ℓ] = Y[ℓ] do
5:          (ℓ, s, p) ← UpdateMS(Y, ℓ, s, p)
        // ℓ = lcp(X_i, Y)
6:      if ℓ = m then
7:          S ← S ∪ {i}
        // MS(Y[0..ℓ)) = Y[s..ℓ) and p = per(Y[s..ℓ))
8:      if p ≤ ℓ/3 and Y[0..s) = Y[p..p + s) then    // per(Y[0..ℓ)) = p
9:          i ← i + p
10:         ℓ ← ℓ − p
11:     else    // per(Y[0..ℓ)) > ℓ/3
12:         i ← i + ⌊ℓ/3⌋ + 1
13:         (ℓ, s, p) ← (0, 0, 0)
14: return S
```

Figure 2. The main procedure of the simplified Crochemore's algorithm.

Lemma 6 *Let* $\mathsf{Y} = \mathsf{P}\mathsf{A}^k\mathsf{B}$ *where* $\mathsf{MS}(\mathsf{Y}) = \mathsf{A}^k\mathsf{B}$ *and* $|\mathsf{B}| < |\mathsf{A}| = p := per(\mathsf{MS}(\mathsf{Y}))$. *Then:*

1. $|\mathsf{P}| < per(\mathsf{Y})$
2. $per(\mathsf{Y}) = per(\mathsf{MS}(\mathsf{Y}))$ *iff* P *is a suffix of* A
3. *if* Y *is 3-periodic then* $per(\mathsf{Y}) = per(\mathsf{MS}(\mathsf{Y}))$

Proof. Let $p' = per(\mathsf{Y})$.

1. Otherwise $\mathsf{A}^k\mathsf{B}$ occurs in Y p' positions earlier, thus is not a maximal suffix.

2. A prefix of Y of length $|\mathsf{P}| + p'$ has a border of length $|\mathsf{P}|$. If $p' = p$ position $|\mathsf{P}| + p$ coincides with the end of A.

The opposite implication follows from the definition of a period.

3. Clearly $p \le p'$ as $\mathsf{A}^k\mathsf{B}$ is a factor of Y. Suppose $p < p'$ and observe that $\mathsf{A}^k\mathsf{B}$ has periods p and p'. Moreover, $3p' \le |\mathsf{Y}|$ and $|\mathsf{P}| < p'$ imply $|\mathsf{A}^k\mathsf{B}| > 2p' > p + p'$ hence from Lemma 1 $\mathsf{A}^k\mathsf{B}$ has also period $p'' := \gcd(p, p')$. But $\mathsf{A}^k\mathsf{B}$ contains an occurrence of $\mathsf{Y}[0..p')$ as a factor thus $\mathsf{Y}[0..p')$ has period $p'' < p'$ and so (since $p'' \mid p'$) the whole Y as well, contradicting the definition of p'.

We immediately obtain the following result (for Y as in Lemma 6).

Corollary 7 Y *is 3-periodic iff* $p \le |\mathsf{Y}|/3$ *and* P *is a suffix of* A.

The pseudo-code of the matching procedure is given in Fig. 2. After computing $\ell = \mathsf{lcp}(\mathsf{X}_i, \mathsf{Y})$ we test if $\mathsf{Y}[0..\ell)$ is 3-periodic using Corollary 7. If it is not, we can safely set $q := \lfloor \ell/3 \rfloor$ and $\ell := 0$. Otherwise, from Lemma 6, we know that $p := per(\mathsf{MS}(\mathsf{Y}[0..\ell))) = per(\mathsf{Y}[0..\ell))$ thus we set $q := p$ and decrease the match length ℓ by p, because the definition of the period implies that we can skip the first $\ell - p$ characters when computing $\mathsf{lcp}(\mathsf{X}_{i+p}, \mathsf{Y})$.

However, now the problem is obtaining the starting position of the maximal suffix of $\mathsf{Y}[0..\ell - p)$ and its shortest period. As explained in the next Lemma, it turns out

that both the starting position and the shortest period of the new maximal suffix stay the same.

This is in contrast with the original Crochemore's algorithm, where 3 cases are considered when performing the shift, each with more involved formulas expressing maximal possible shifts. It results in a tight upper bound on the number of comparisons, but at the cost of intricate complexity analysis and the need for more formal statements.

Lemma 8 *Assume* Y *is a 3-periodic string of length* ℓ*. Let* $\mathsf{MS}(\mathsf{Y}) = \mathsf{Y}_s$ *and* $per(\mathsf{Y}_s) = p$*. Then for* $\mathsf{Y}' := \mathsf{Y}[0..\ell - p)$ *we have* $\mathsf{MS}(\mathsf{Y}') = \mathsf{Y}'_s$ *and* $per(\mathsf{Y}'_s) = p$*.*

Proof. Suppose $\mathsf{MS}(\mathsf{Y}') = \mathsf{Y}'_{s'}$ for $s' \neq s$. The only case that does not immediately yields $\mathsf{Y}_{s'} > \mathsf{Y}_s$ (contradicting $\mathsf{MS}(\mathsf{Y}) = \mathsf{Y}_s$) is when $s' < s$ and Y'_s is a prefix of $\mathsf{Y}'_{s'}$. It is also its suffix, thus $\mathsf{Y}'_{s'}$ has a period $s - s'$. It also has period p and inequalities $|\mathsf{Y}'| \geq 2p$, $s' < s < p$ (recall Lemma 6(1)) imply $|\mathsf{Y}'_{s'}| \geq 2p - s' > p + (s - s')$, thus from Lemma 1 $\mathsf{Y}'_{s'}$ has period $p' := \gcd(p, s - s') < p$. But $\mathsf{Y}'_{s'}$ contains an occurrence of $\mathsf{Y}[0..p)$ hence $\mathsf{Y}[0..p)$ must also have period p', and since $p' \mid p$ the whole Y as well, a contradiction.

Clearly $per(\mathsf{Y}'_s) \leq p$, as Y'_s is a factor of Y. It cannot be $p' := per(\mathsf{Y}'_s) < p$ because then $\mathsf{Y}'_s[p'..p)$ is a border of $\mathsf{Y}_s[0..p)$ which is impossible by Lemma 4.

Theorem 9 `Match` *runs in* $\mathrm{O}(n + m)$ *time and uses* $\mathrm{O}(1)$ *extra space.*

Proof. Clearly only a constant number of integer variables are used throughout the computation and neither the text nor the pattern are modified.

Each step of the while loop in line 3 increases the value of the non-decreasing expression $3i + \ell$, thus it is executed at most $3n + m = \mathrm{O}(n + m)$ times.

The total cost of `UpdateMS` is bounded by the total increase of s (Observation 5). The maximal value of s is $m - 1$ and it can only decrease in line 13. But since $s < \ell$ and the decrease is always followed by increasing i by $\lfloor \ell/3 \rfloor + 1 > s/3$, s can overall increase by at most $3n + m = \mathrm{O}(n + m)$.

Finally, we divide the checks $\mathsf{Y}[0..s) = \mathsf{Y}[p..p+s)$ into two groups. If the condition in line 8 evaluates to true we have $s < per(\mathsf{Y}[0..\ell)) = p$ (see Lemma 6) and i is immediately increased by p (line 9), thus the total cost of such checks is $\mathrm{O}(n)$. Otherwise i is incremented by $\lfloor \ell/3 \rfloor + 1 > s/3$ (line 12). The maximal value of i is n, thus such checks overall cost at most is $3n = \mathrm{O}(n)$.

4 Extensions

4.1 Longest Prefix Matching

We search a pattern Y inside X and keep track of the length $\ell_{\max}$ of the longest matching prefix of Y found so far. The pseudo-code, which is a straightforward modification of the `Match` procedure is given in Fig. 3. During the computation we maintain a set of text positions $\mathcal{S}$ such that $j \in \mathcal{S}$ iff $\mathsf{lcp}(\mathsf{X}_j, \mathsf{Y}) = \ell_{\max}$.

Algorithm `LongestPrefixMatch`$(\mathsf{X}, n, \mathsf{Y}, m)$

Input: strings $\mathsf{X}[0..n)$ (text) and $\mathsf{Y}[0..m)$ (pattern).
Output: $\ell_{\max} = \max_{i \in [0..n)} \mathsf{lcp}(\mathsf{X}_i, \mathsf{Y})$ and $\mathcal{S} = \{j \in [0..n) \mid \mathsf{lcp}(\mathsf{X}_j, \mathsf{Y}) = \ell_{\max}\}$.

```
1:  S ← ∅
2:  i ← p ← s ← ℓ ← ℓ_max ← 0
3:  while i < n do
4:      while i + ℓ < n and ℓ < m and X[i + ℓ] = Y[ℓ] do
5:          (ℓ, s, p) ← UpdateMS(Y, ℓ, s, p)
6:      if ℓ > ℓ_max then
7:          S ← {i}
8:          ℓ_max ← ℓ
9:      elsif ℓ = ℓ_max then
10:         S ← S ∪ {i}
11:     if p ≤ ℓ/3 and Y[0..s) = Y[p..p + s) then
12:         i ← i + p; ℓ ← ℓ − p
13:     else
14:         i ← i + ⌊ℓ/3⌋ + 1; (ℓ, s, p) ← (0, 0, 0)
15: return (ℓ_max, S)
```

Figure 3. The basic algorithm solving Longest Prefix Matching problem.

Theorem 10 *The algorithm* `LongestPrefixMatch` *solves the Longest Prefix Matching problem in linear time.*

Proof. The time complexity follows from Theorem 9.

To prove its correctness, observe that after line 10 we have $\ell_{\max} \geq \ell$. The shift that follows is not longer than $per(\mathsf{Y}[0..\ell))$, so from Observation 2 all positions j that we skip satisfy $\mathsf{lcp}(\mathsf{X}_j, \mathsf{Y}) < \ell \leq \ell_{\max}$, i.e., we only omit the candidates for $\ell_{\max}$ that would not change its value nor end up in $\mathcal{S}$.

Note that the peak size of set $\mathcal{S}$ can be much larger than the final output. For instance when $\mathsf{X} = a^q b$ and $\mathsf{Y} = ab$ the size of $\mathcal{S}$ reaches $q - 1$ but the final $\mathcal{S}$ satisfies $|\mathcal{S}| = 1$.

It is possible to get rid of this overhead as follows. First run `LongestPrefixMatch` but only record the length $\ell_{\max}$. Then, in the second run, collect exclusively the elements on the final set $\mathcal{S}$, which can now be easily recognized. We have proved the following

Theorem 11 *It is possible to solve the Longest Prefix Matching problem in linear time and using only constant extra space in addition to the input and the output.*

4.2 Sparse Longest Prefix Matching

Let $\mathcal{P}$ be the ascending sequence of text positions given in addition to the text X and the pattern Y. In order to solve the sparse variant of the problem, we proceed exactly the same as in the basic version, but only execute lines 4-10 if $i \in \mathcal{P}$. We call this modified algorithm `SparseLongestPrefixMatch`.

Theorem 12 *The algorithm* `SparseLongestPrefixMatch` *solves the Sparse Longest Prefix Matching problem in linear time.*

Proof. The condition $i \in \mathcal{P}$ can be checked in constant time since i never decreases and the elements of $\mathcal{P}$ are given in ascending order. The analysis from Theorem 9 applies to the rest of the algorithm.

In order to prove its correctness observe that whenever we are about to execute lines 4-10 the condition $\ell \leq \ell_{\max}$ is satisfied, even if $i \notin \mathcal{P}$. This is because ℓ increases *only* for positions $i \in \mathcal{P}$ and any such increase is immediately recorded in lines 6-10. Therefore the argument from Theorem 10 also applies here, i.e., the positions in the text that are not inspected would never contribute to the answer.

An identical technique as for `LongestPrefixMatch` can be applied to reduce the memory overhead caused by the large peak size of $\mathcal{S}$ yielding

Theorem 13 *It is possible to solve the Sparse Longest Prefix Matching problem in linear time and using only constant extra space in addition to the input and the output.*

References

1. R. S. Boyer and J. S. Moore: *A fast string searching algorithm.* Communications of the ACM, 20(10) 1977, pp. 762–772.
2. M. Crochemore: *String-matching on ordered alphabets.* Theoretical Computer Science, 92 1992, pp. 33–47.
3. M. Crochemore and W. Rytter: *Sqares, cubes, and time-space efficient string searching.* Algorithmica, 13(5) 1995, pp. 405–425.
4. J.-P. Duval: *Factorizing words over an ordered alphabet.* Journal of Algorithms, 4(4) 1983, pp. 363–381.
5. S. Faro and T. Lecroq: *The exact online string matching problem: A review of the most recent results.* ACM Computing Surveys, 45(2) 2013, article 13.
6. P. Ferragina and G. Manzini: *On compressing the textual web*, in WSDM, ACM, 2010, pp. 391–400.
7. N. J. Fine and H. S. Wilf: *Uniqueness theorems for periodic functions.* Proceedings of the American Mathematical Society, 16(1) 1965, pp. 109–114.
8. T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. J. Puglisi: *A faster grammar-based self-index*, in LATA, vol. 7183 of LNCS, Springer, 2012, pp. 240–251.
9. T. Gagie, P. Gawrychowski, and S. J. Puglisi: *Faster approximate pattern matching in compressed repetitive texts*, in ISAAC, vol. 7074 of LNCS, Springer, 2011, pp. 653–662.
10. Z. Galil and J. Seiferas: *Time-space optimal string matching.* Journal of Computer and System Sciences, 26 1983, pp. 280–294.
11. Z. Galil and J. I. Seiferas: *Saving space in fast string-matching.* SIAM Journal on Computing, 9(2) 1980, pp. 417–438.
12. C. Hoobin, S. J. Puglisi, and J. Zobel: *Relative Lempel-Ziv factorization for efficient storage and retrieval of web collections.* Proceedings of the VLDB Endowment, 5(3) 2011, pp. 265–273.
13. J. Kärkkäinen, D. Kempa, and S. J. Puglisi: *Lempel-Ziv parsing in external memory.* Manuscript, `http://arxiv.org/abs/1307.1428`, 2013.
14. J. Kärkkäinen, D. Kempa, and S. J. Puglisi: *Lightweight Lempel-Ziv parsing*, in SEA, vol. 7933 of LNCS, Springer, 2013, pp. 139–150.
15. J. Kärkkäinen, D. Kempa, and S. J. Puglisi: *Linear time Lempel-Ziv factorization: Simple, fast, small*, in CPM, vol. 7922 of LNCS, Springer, 2013, pp. 189–200.
16. D. Kempa and S. J. Puglisi: *Lempel-Ziv factorization: simple, fast, practical*, in ALENEX, SIAM, 2013, pp. 103–112.
17. D. Knuth, J. H. Morris, and V. Pratt: *Fast pattern matching in strings.* SIAM Journal on Computing, 6(2) 1977, pp. 323–350.
18. S. Kreft and G. Navarro: *Self-indexing based on LZ77*, in CPM, vol. 6661 of LNCS, Springer, 2011, pp. 41–54.
19. J. H. Morris, Jr and V. R. Pratt: *A linear pattern-matching algorithm*, Report 40, University of California, Berkeley, 1970.
20. J. Ziv and A. Lempel: *A universal algorithm for sequential data compression.* IEEE Transactions on Information Theory, 23(3) 1977, pp. 337–343.