Words distinguished by their subwords*

Extended Abstract

Imre Simon†

August 31, 2003

Dedicated to the memory of my brother, Janos Albert Simon

Abstract

We present an $O(|A|(|x_1|+|x_2|))$ algorithm to find a shortest subword which divides x_1 iff it does not divide x_2, where x_1 and x_2 are words over the alphabet A. Based on the length of such a word an ultra-metric distance on A^* can be defined. The algorithm can be transformed to work in time and space $O(|A|+|x_1|+|x_2|)$.

1 Distinguishing words by their subwords

Let A be a finite alphabet. Given words y, x_1 and x_2 in A^* we say that y *distinguishes* x_1 and x_2 if y is a subword of exactly one of x_1 and x_2. Here, we say that y is a subword of x, and write $y|x$, if there exist words $y_1, y_2, \ldots, y_n, x_0, x_1, \ldots, x_n$ in A^*, for some $n \geq 1$, such that

$$y = y_1 y_2 \cdots y_n \quad \text{and} \quad x = x_0 y_1 x_1 y_2 x_2 \cdots y_n x_n.$$

That is to say, sequence y is a subsequence of sequence x. We might also say that y divides x.

*This work was supported by research grants from CNPq.

†Instituto de Matemática e Estatística, Universidade de São Paulo, 05508 São Paulo, SP, Brasil. E-mail: <is@ime.usp.br>

Clearly, there exist a word distinguishing x_1 and x_2 iff x_1 and x_2 are distinct. For an example, let us consider the alphabet $A = \{ a, b, c, d \}$ and let x_1 and x_2 be the words:

$$x_1 = bcaca\ cadca\ babab\ dacba$$

$$x_2 = bacca\ dcabb\ cbcab\ adabc$$

Then, $y = dacbac$, of length 6 distinguishes them. Indeed, $y|x_2$ but $y \not| x_1$. However, there are shorter words distinguishing them. For example, for $y = aacdd$ we have that $y|x_1$ but $y \not| x_2$.

The purpose of this paper is to present an algorithm which finds a *shortest* word which distinguishes two given distinct words. The algorithm we present will have time and space complexities $O(|A|(|x_1|+|x_2|))$. Introducing suitable data structures it can be transformed to work in time and space $O(|A|+|x_1|+|x_2|)$ which makes it suitable for working with very long words, such as the human genome, for instance, whose length is of the order of 10^{10}.

The existence of such a fast algorithm might be surprising since there might be exponentially many subwords dividing a word of a given length. For instance, any word of length m divides $(abcd)^m$. Also, there are many papers constructing the automaton accepting all the subwords of a given word of length n in time and space $O(|A||x|)$. Hence, our final algorithm works faster than these constructions.

A (not so) naive algorithm to compute a shortest distinguisher of x_1 and x_2 can easily be made to work in time $O(|A||x_1||x_2|)$. Indeed, one would use the above mentioned algorithms to construct the automata $\mathcal{A}_1$ and $\mathcal{A}_2$ accepting, respectively, all the subwords of x_1 and x_2. It would suffice to compute the direct product of these automata and find the shortest word in $|\mathcal{A}_1|\Delta|\mathcal{A}_2|$. Here, we denote by $|\mathcal{A}|$ the behavior of automaton $\mathcal{A}$. In a remarkable paper [4] Emmanuelle Garel made a construction which efficiently finds all the shortest words distinguishing two given distinct words.

Most of the basic ideas of our algorithm can be found in the proof of Theorem 6.2.6 of [6]. Indeed, the algorithm presented here is just a careful following of the construction given in that proof. The result there says that if words x_1 and x_2 have the same subwords of length at most m, for some m, then x_1 and x_2 can be merged into a new word z, without introducing any new subwords of length up to m. Indeed, our algorithm constructs such a word z for the greatest possible value of m.

In our example, the largest m for which x_1 and x_2 have the same subwords is $m = 3$ and a shortest word distinguishing x_1 and x_2 is $y = dccc$. A common multiple, z, of x_1 and x_2, without new subwords of length up to 3, would be

$$z = bcacc\ acadc\ abbcb\ cabab\ dacbc\ a.$$

It can be readily verified that z is a common superword of x_1 and x_2, but verifying that its subwords of length at most $m = 3$ are exactly the same as those of x_1 (or of x_2) might not be so immediate.

Let x_1 and x_2 be two words in A^* and in case they are different, let y be a shortest word distinguishing them. Let us define

$$\delta(x_1, x_2) = \begin{cases} \infty & \text{if } x_1 = x_2, \\ |y| - 1 & \text{otherwise.} \end{cases}$$

In [6] it is proved that $\theta(x_1, x_2) = 2^{-\delta(x_1,x_2)}$ is an ultra-metric distance on A^*. In terms of δ this just means that for any x_1, x_2 and x_3 in A^*

$$\delta(x_1, x_2) \geq \min\{\, \delta(x_1, x_3), \delta(x_2, x_3)\,\},$$

which is just the expression of the triangular inequality in the case of an ultra-metric distance. Thus, our algorithm can be seen as computing this distance in A^*.

Besides making use of the ideas in [6] for the justification of its correctness the algorithm obtains the complexity $O(|A|(|x_1|+|x_2|))$ through the following main features.

1. Replacement of the recursive structure implied by the proof of Theorem 6.2.6 by an iterative one.

2. The use of a more efficient algorithm to compute a particular case: finding a shortest word distinguishing u and ua, where $u \in A^*$ and $a \in A$. This same algorithm is also used dually for the words au and u. Indeed, these simpler cases are the basic building blocks of our algorithm. They correspond to the cases of Lemma 6.2.12 and Proposition 6.2.15 of [6] but we use a different method to compute $\delta(u, ua)$.

3. The careful computation and storage of certain tables of size at most $|A|(|x_1| + |x_2|)$, some of which are computed by pre-processing x_1 and

x_2 and whose use will guide the construction of the merging of x_1 and x_2 into z. Two of these tables are just the subword automata of the reverses of x_1 and x_2.

4. Representation of certain families of subwords of x_1 and x_2 through the use of a list structure; this permits the storage of suitable sets of n words of length at most n using only n pointers. These pointers are stored in the tables mentioned in the previous item.

We note that ours is not an on-line algorithm. Actually, different phases of the algorithm run through the input words in opposite directions and this dual feature seems to be essential to our methods. Indeed, the design of an efficient on-line algorithm, if at all possible, might require entirely new ideas.

2 The algorithm

The algorithm is completely specified in a number of procedures given in Algorithms 2.1, 2.2, 2.3, 2.4 and 2.5.

We silently assume that the alphabet can be efficiently enumerated, indexed and accessed, just as if it were the set $\{1, 2, \ldots, k\}$, for instance.

Algorithm 2.1 Computing a shortest distinguisher for x_1 and x_2

Distinguisher(x_1, x_2)
$\triangleright$ Preparing the leftist data structures
$\text{Ld}_1 \leftarrow$ LeftDistinguisher(x_1, x_2)
$\text{Ld}_2 \leftarrow$ LeftDistinguisher(x_2, x_1)
$\triangleright$ Merging x_1 and x_2 into z
$(z, \text{Rd}, m, jm, psm, (im, am)) \leftarrow$ Merge(x_1, x_2)
if $m = \infty$ **then** $\triangleright$ $x_1 = x_2$, they can not be distinguished
else $yr \leftarrow$ Collect$(\text{Rd}, (jm - 1, z[jm]))$
$yl \leftarrow$ Collect$(\text{Ld}_{psm}, (im, am))$
$y \leftarrow \text{reverse}(yr).z[jm].yl$
$\triangleright$ We just collected a shortest distinguisher in y, $|y| = m + 1$
$|y| \leftarrow m + 1$
return (m, y)

Algorithm 2.1 is the "main procedure". It specifies the major steps to be executed by the other procedures. Initially it constructs the tables Ld_1 and

$\textsc{Ld}_2$ using Algorithm 2.2. Next, it merges x_1 and x_2 into z and computes some auxiliary data using Algorithm 2.3. Finally, using Algorithm 2.5 it collects the left and right factors of the shortest distinguisher computed.

Throughout the program a word x is represented by a vector of letters, say $x[i]$, and we assume that the length of x is given (and computed when it is the case) in a variable called $|x|$.

The value m returned by Algorithm 2.1 is just $\delta(x_1, x_2)$ as defined earlier. And in case $m \neq \infty$ a shortest distinguisher y of length $m+1$ is also returned.

Algorithm 2.2 Constructing the left distinguisher matrix LD of a word x with embeddings in word y.

```
LEFTDISTINGUISHER(x, y)
▷ We compute the subword automaton of the reverse of y
for a ∈ A do
        AUTO[0, a] ← DEADSTATE
        AUTO[DEADSTATE, a] ← DEADSTATE
for i from 1 to |y| do
        for a ∈ A do
                if y[i] = a then
                        AUTO[i, a] ← i − 1
                else    AUTO[i, a] ← AUTO[i − 1, a]
▷ We compute the Left Distinguisher matrix LD of x
for a ∈ A do
        LD[|x|, a] ← (0, NIL, |y|)
for i from |x| to 1 do
        for a ∈ A do
                (l, n, s) ← LD[i, x[i]]
                if a = x[i] or length(LD[i, a]) > 1 + l then
                        LD[i − 1, a] ← (1 + l, (i, x[i]), AUTO[s, x[i]])
                else    LD[i − 1, a] ← LD[i, a]
return LD
```

Algorithm 2.2 computes the main tables driving the merging process. First, the subword automaton AUTO of the reverse of word y is computed. This is a table of size $|A|(|y|+2)$ and it will only be used inside this procedure. The states $\{ 0, 1, 2, \ldots, |y| \}$ of the automaton correspond to the factorization $y = uv$, with u of length $\{ 0, 1, 2, \ldots, |y| \}$ respectively. The remaining state is a dead state.

The second half of Algorithm 2.2 computes another table, $\textsc{Ld}$, of size $4|A|(|x|+1)$. A typical element is $\textsc{Ld}[i,a]$ where $i \in \{0,1,2,\ldots,|x|\}$ describes a factorization of x, as described above and a is a letter in the alphabet. The typical value of an element is a three-tuple:

$$\textsc{Ld}[i,a] = (l,(i',a'),s).$$

$\textsc{Ld}[i,a]$ represents a shortest distinguisher y of v and av, where $x = uv$ is the factorization of x with $|u| = i$. Such a distinguisher always begins with the letter a and is followed by a word of length l which begins in $\textsc{Ld}[i',a']$. We will refer to this word of length l associated to $\textsc{Ld}[i,a]$ as the word stored in $\textsc{Ld}[i,a]$. The procedure of Algorithm 2.5 is used exactly to recover the word stored in $\textsc{Ld}[i,a]$. Whenever $l = 0$ the word stored in $\textsc{Ld}[i,a]$ is the empty word and the pointer (i',a') is substituted by NIL.

Finally, the value s represents a factorization $y = uv$ of the word y, where $|u| = s$. This s is the longest prefix of y for which v still contains an occurrence of the word stored in $\textsc{Ld}[i,a]$. Theses values s are critically used in the procedure RACE in Algorithm 2.4. Note that the values of s are computed by using the subword automaton previously computed.

Algorithm 2.3 is a major step in the computation of a shortest distinguisher. It is based on the construction of Theorem 6.2.11 in [6]. The word z is computed by a classical merging algorithm applied to x_1 and x_2, read from the left to the right. The interesting case is that of a "Mismatch". This case corresponds to an application of Lemma 6.2.12 of [6] and it is basically handled by Algorithm 2.4.

The procedure MERGE includes the computation of a distinguisher structure $\textsc{Rd}$ dual to the structures computed by Algorithm 2.2 and there described. This time we proceed from left to right and compute shortest distinguishers of u and ua, for prefixes u of z and letters $a \in A$. The data is stored in $\textsc{Rd}[|u|,a]$ in a fashion similar to $\textsc{Ld}$. The word stored in $\textsc{Rd}[i,a]$ is retrieved by the COLLECT procedure on Algorithm 2.5 and its reverse, followed by a is a shortest distinguisher of u and ua, for u a length i prefix of z. The use of the structure $\textsc{Rd}$ does not require the third component s of the elements $\textsc{Rd}[i,a]$. Thus, we put $s =$ NIL in every case.

We would like to note here that the storage of most of the data in $\textsc{Rd}$ is unnecessary for the algorithm. We decided to include it here in its entirety because it does not compromise the complexity of the algorithm but on the other hand it simplifies (a little) the explanation because we have

Algorithm 2.3 Merging words x_1 and x_2 into z without creating new short subwords

```
Merge(x1, x2)
(k1, k2, j, m) ← (0, 0, 0, ∞)
for a ∈ A do
        Rd[0, a] ← (0, nil, nil)
while k1 < |x1| or k2 < |x2| do
        j ← j + 1
        if k1 < |x1| and k2 < |x2| then
                (b1, b2) ← (x1[k1 + 1], x2[k2 + 1])
                if b1 = b2 then
                        (pz, k2, Case) ← (1, k2 + 1, Match)
                else    Case ← Mismatch
                        (pz, ps, (i′, a′))) ← Race(k1, b2, k2, b1)
        else    Case ← Singleton
                if k1 = |x1| then
                        pz ← 2
                else    pz ← 1
                ▷ (i′, a′) is a pointer in Ld_ps
                (ps, (i′, a′)) ← (pz, (|x_ps|, z[j]))
        (k_pz, z[j]) ← (k_pz + 1, x[pz][k_pz])
        for a ∈ A do
                (l, n, s) ← Rd[j − 1, z[j]]
                if a = z[j] or length(Rd[j − 1, a]) > 1 + l then
                        Rd[j, a] ← (1 + l, (j − 1, z[j]), nil)
                else    Rd[j, a] ← Rd[j − 1, a]
        if Case ≠ Match then
                m′ ← length(Rd[j − 1, z[j]]) + length(Ld_ps[i′, a′])
                if m = ∞ or m′ < m then
                        (m, jm, psm, (im, am)) ← (m′, j, ps, (i′, a′))
|z| ← j
return (z, Rd, m, jm, psm, (im, am))
▷ If m = ∞ then jm, psm, im and am are undefined
```

dual computations for LD and RD which can be handled and understood simultaneously.

In each non-match case of the MERGE procedure we have a value m' of m. This value is constituted by the sum of lengths of a word stored in RD and one stored in an appropriate LD. The corresponding word is mounted using COLLECT and inserting in between the appropriate letter of z. Confront the construction with Lemma 6.2.13 of [6]. During the merging procedure we compute the least value of m and the data referent to this value is preserved in the variables m, jm, psm and (im, am). Note that (i', a') is a pointer in LD_{ps} which is necessary to recover the word stored in $\mathrm{LD}_{ps}[i', a']$.

Algorithm 2.4 An auxiliary procedure which confronts x_1 and x_2 in the case of a mismatch

RACE(k_1, b_2, k_2, b_1)

if length$(\mathrm{LD}_1[k_1, b_2]) \leq$ length$(\mathrm{LD}_2[k_2, b_1])$ **then**
    $(ps, pl) \leftarrow (1, 2)$
else $(ps, pl) \leftarrow (2, 1)$
▷ $\mathrm{LD}_{ps}[k_{ps}, b_{pl}]$ is the left distinguisher which won the race
$(i', a') \leftarrow (k_{ps}, b_{pl})$
if next$(\mathrm{LD}_{ps}[k_{ps}, b_{pl}]) =$ NIL **or** suffix$(\mathrm{LD}_{ps}[k_{ps}, b_{pl}]) \geq 1 + k_{pl}$ **then**
    $pz \leftarrow pl$
else $pz \leftarrow ps$
return $(pz, ps, (i', a'))$ ▷ (i', a') is a pointer in LD_{ps}

Algorithm 2.4 is an implementation of Lemma 6.2.14 of [6]. It is a crucial part of the merging procedure and it is the one deciding which letter to promote in the case of a "mismatch".

Finally, Algorithm 2.5 is used to retrieve the words stored in the elements of the distinguisher structures.

A few words about the complexity of the algorithm are in order. Since every line in the algorithm is surrounded at most by an A-loop and an x-loop, refering respectively to the alphabet and the input words, the complexity is clearly bounded by $O(|A|(|x_1| + |x_2|))$.

Algorithm 2.5 Collecting the word encoded in $\mathrm{D}[i,a]$, where D is a distinguisher matrix LD or RD

COLLECT$(\mathrm{D}, (i,a))$

$\triangleright$ We collect the word y encoded in $\mathrm{D}[i,a]$; with $|y| = \mathrm{length}(\mathrm{D}[i,a])$
$(j,k) \leftarrow (1, \mathrm{length}(\mathrm{D}[i,a]))$
while $k > 0$ **do**
    $(l,(i,a),s) \leftarrow \mathrm{D}[i,a]$ $\triangleright$ We must have $l = k$
    $y[j] \leftarrow a$
    $(j,k) \leftarrow (j+1, k-1)$
$|y| \leftarrow j-1$
return y

3 About the $O(|A| + |x_1| + |x_2|)$ version

We will not present this algorithm here in detail, but expect to do it shortly. It is obtained through a number of transformations. The basic technique is an inversion procedure of the matrices AUTO, LD and RD which are the ones using time and space $O(|A||x|)$. The first observation is that all these matrices are "sparse", that is to say most of their elements are repetitions of previous elements. This can be seen in the algorithms computing these structures where lines 9 and 18 in 2.2 and line 22 in 2.3 reproduce earlier elements. Indeed, the structures LD and RD are representations of trees. It can be proved that the number of nodes in these trees is bounded by a linear function of $|A| + |x|$, where $|x|$ and $|A|$ are the respective dimensions of the structures we use here.

A similar situation occurs with the subword automata AUTO. The alert reader will note that in this automaton, in general, many consecutive a-edges have the same endpoint. Indeed, from this perspective there are exactly $|x|$ classes of edges. This observation allows the use of appropriate inversion techniques to represent and traverse these structures. The corresponding economies allow the reduction of the overall complexity of the algorithm to $O(|A| + |x_1| + |x_2|)$.

4 Some historical remarks

This work was done in the course of the last 30 years. Indeed, the first algorithm to find a shortest distinguisher, even if not phrased in these terms, is part of the author's Ph.D. Dissertation, written in 1972. The algorithm appears in the Chapter characterizing piecewise testable events (called events in γ_1 at that time) as those whose syntactic monoid is finite and $\mathcal{J}$-trivial. An "efficient" algorithm is explicitly mentioned there, in quotes. The complexity of that algorithm was never even computed, it must have been a polynomial of degree 3 or 4.

Later on, in 1975, a new, simpler proof of the $\mathcal{J}$-trivial theorem was found by the author. This proof uses the common multiple z which is crucial in our algorithm.

The first appearance of our algorithm, pretty much in the form it is given here, occurred during a NATO Summer School held in Maratea, in 1984. Unforeseeable circumstances prevented us from publishing this work at that time. Later on, in 1986, we proceeded to the improvement of the complexity which is signaled here. We lectured about the improved version in Paris in 1986 and mentioned our algorithm in a survey paper, written in 1989, about the longest common subword of two words, used in file differencing algorithms and in alignments of biological sequences, among many other uses. But, again, the results were not published. The author hopes that this time he will have better luck and that the complete version of this paper will soon see the light of the day.

References

[1] R. A. Baeza-Yates. Searching subsequences. *Theoretical Computer Science*, 78(2):363–376, Jan. 1991.

[2] Crochemore and Tronicek. On the size of DASG for multiple texts. In *International Symposium on String Processing and Information Retrieval, SPIRE, LNCS*, volume 9, 2002.

[3] M. Crochemore and Z. Troníček. Directed acyclic subsequence graph for multiple texts. Rapport I.G.M. 99-13, Université de Marne-la-Vallée, 1999.

[4] E. Garel. Minimal separators of two words. In A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, editors, *Combinatorial Pattern Matching, 4th Annual Symposium*, volume 684 of *Lecture Notes in Computer Science*, pages 35–53, Padova, Italy, 2-4 June 1993. Springer.

[5] J.-J. Hébrard and M. Crochemore. Calcul de la distance par les sous-mots. *RAIRO Inform. Théor. Appl.*, 20(4):441–456, 1986.

[6] J. Sakarovitch and I. Simon. Subwords. In M. Lothaire, editor, *Combinatorics on Words*, pages 105–144. Addison-Wesley, Reading, 1983. Encyclopaedia of Mathematics, volume 17.

[7] I. Simon. *Hierarchies of Events with Dot-Depth One*. PhD thesis, University of Waterloo, Waterloo, Ont., Canada, 1972. Department of Applied Analysis and Computer Science.

[8] I. Simon. Piecewise testable events. In H. Brakhage, editor, *Automata Theory and Formal Languages, 2nd GI Conference*, pages 214–222, Berlin, 1975. Springer-Verlag. Lecture Notes in Computer Science, 33.

[9] I. Simon. Sequence comparison: some theory and some practice. In M. Gross and D. Perrin, editors, *Electronic Dictionaries and Automata in Computational Linguistics*, pages 79–92, Berlin, 1989. Springer-Verlag. Lecture Notes in Computer Science, 377.

[10] Z. Troníček. Common subsequence automaton. In *International Conference on Implementation and Application of Automata 2002*, Tours, France, 2002.