# Efficient Text Fingerprinting Via Parikh Mapping *

Amihood Amir[†]
Bar-Ilan University
and
Georgia Tech

Alberto Apostolico[‡]
Università di Padova
and
Purdue University

Gad M. Landau[§]
Haifa University
and
Polytechnic University

Giorgio Satta[¶]
Università di Padova

**Abstract**

We consider the problem of fingerprinting text by sets of symbols. Specifically, if $S$ is a string, of length $n$, over a finite, ordered alphabet $\Sigma$, and $S'$ is a substring of $S$, then the *fingerprint* of $S'$ is the subset $\phi$ of $\Sigma$ of precisely the symbols appearing in $S'$.

In this paper we show efficient methods of answering various queries on fingerprint statistics. Our preprocessing is done in time $O(n|\Sigma|\log n \log|\Sigma|)$ and enables answering the following queries:

1. Given an integer $k$, compute the number of distinct fingerprints of size $k$ in time $O(1)$.
2. Given a set $\phi \subseteq \Sigma$, compute the total number of distinct occurrences in $S$ of substrings with fingerprint $\phi$ in time $O(|\Sigma|\log n)$.

**Key Words:** Design and analysis of algorithms, combinatorial algorithms on words,

---

*This research was performed during exchange visits conducted, respectively, by the first and third authors at the University of Padova, and by the second author at the Universities of Bar-Ilan and Haifa, as part of an Israel-Italy exchange scientist grant jointly funded by the Israel Ministry of Science and the National Research Council of Italy.

†Department of Mathematics and Computer Science, Bar-Ilan University, 52900 Ramat-Gan, Israel, (972-3)531-8770, amir@cs.biu.ac.il. Partially supported by NSF grant CCR-01-04494, BSF grant 96-00509, and an Israel-Italy exchange scientist grant.

‡Dipartimento di Elettronica e Informatica, Università di Padova, Via Gradenigo 6/A, 35131 Padova, Italy, and Department of Computer Sciences, Purdue University, Computer Sciences Building, West Lafayette, IN 47907, USA, (39-049)827-7728 ; axa@cs.purdue.edu. Work supported in part by NSF Grant CCR-9700276, by MURST under project *PRIN: BioInformatica e Ricerca Genomica*, by the Univeristy of Padova under project *Development of Novel Pattern Discovery Algorithms and Software*, and by an Israel-Italy exchange scientist grant.

§Department of Computer Science, Haifa University, Haifa 31905, Israel, phone: (972-4) 824-0103, FAX: (972-4) 824-9331; Department of Computer and Information Science, Polytechnic University, Six MetroTech Center, Brooklyn, NY 11201-3840; email: landau@poly.edu; partially supported by NSF grants CCR-9610238, and CCR-0104307, by NATO Science Programme grant PST.CLG.977017, by the Israel Science Foundation grants 173/98 and 282/01, by the FIRST Foundation of the Israel Academy of Science and Humanities, and by IBM Faculty Partnership Award.

¶Dipartimento di Elettronica e Informatica, Università di Padova, Via Gradenigo 6/A, 35131 Padova, Italy, (39-049)827-7831; satta@dei.unipd.it; Work supported in part by MURST under project *PRIN: BioInformatica e Ricerca Genomica* and by University of Padova, under project *Sviluppo di Sistemi ad Addestramento Automatico per l'Analisi del Linguaggio Naturale*.

# 1 Introduction

Automatic rule induction and rule discovery techniques have been applied since the early 50's in several fields where data from physical observations were available in electronic form [5, 6]. Rule discovering is mainly concerned with the problem of inferring generalizations of the input data, to be further exploited by automatic classification systems or by predictive models of the phenomena of interest. In this paper we investigate a problem that arises in the context of the induction of rules for certain natural language processing tools called part-of-speech taggers, as discussed below.

As is well-known, natural languages have a high degree of lexical ambiguity, meaning that most common words may belong to several lexical categories, as for instance Noun, Verb, Adjective, etc. Despite this ambiguity, native language speakers can almost always determine a unique, *intended* lexical category for each word occurrence in natural language texts. To give a simple example, the English word "rule" belongs to both lexical categories Verb and Noun. But in the context of the sentence "This is a rule", the correct classification for the word "rule" is only one, namely Noun. In natural language processing applications, this classification task can be automatically performed by tools that are called part-of-speech taggers.

A common solution in the design of part-of-speech taggers is to exploit sets or cascades of rules that are automatically induced from classification examples. Coming back to our example sentence "This is a rule", a nice rule that we could adopt is to choose the category Noun whenever the preceding word has already been classified with the category Determiner (this is the case for the word "a" in our sentence, when we process the input form left to right). But how would we generalize the observed rule to work also for sentences like "This is a good rule", or "This is a slightly better rule"? In the general case, it turns out that very high accuracy can be achieved by specifying a *set* of lexical categories that are allowed to occur in between some left trigger word and the word under classification. In our example, we could choose the category Noun (given the choice of Verb and Noun) whenever a left triggering word has been classified as Determiner and is followed by words classified by categories in the set {Adjective, Adverb}. The use of sets is particularly effective because very specific category distributions are found in the surrounding context of several lexical categories to be discriminated, while the order of appearance of these surrounding categories is often unpredictable and does not affect the classification accuracy. We refer the interested reader to [2] for more details on the use of sets as constraints in the task of lexical categorization.

In the above perspective, methods for the automatic induction of lexical classification rules must rely on analysis of symbol distributions, more specifically on the statistics of repetitive distributions of the same sets of symbols within some input strings. The problem of interest can be formalized as follows. Let $S$ be a string of $n$ symbols over some alphabet $\Sigma$. We say that a substring $S'$ occurring within $S$ has *fingerprint* $\Sigma' \subseteq \Sigma$ if $\Sigma'$ is the set of all and only those symbols of $\Sigma$ that have at least one occurrence in $S'$. The fingerprint $\Sigma'$ is also called the *alphabet* of $S'$, and denoted by $alph(S')$. We are interested in computing statistics on the various fingerprints observed within $S$, and would like to efficiently construct data structures that allows for fast responses to queries of the following type:

- Given $k$, how many distinct fingerprints of size $k$ exist in $S$?
- Given subset $\phi \in \Sigma$, how many substring occurrences exist in $S$ that have fingerprint $\phi$?

In addition to natural language processing, the above-mentioned problems are of interest in several other domains and applications in which classifications strongly depend on feature *sets* distributions, as opposed to feature *sequences* distributions. In fact, the approach to fingerprint computation presented in this paper solves the more general problem of computing the *Parikh Mapping* of all substrings of a given text string. The Parikh mapping is a morphism from $\Sigma^*$ to the set of ordered $|\Sigma|$-tuples of non-negative integers, that associates with every string $S$ the array $COUNTER[1..|\Sigma|]$ such that $COUNTER[j]$ is the number of occurrences in $S$ of the $j$-th alphabet letter. Parikh mappings enter some fundamental constructions and properties in the theory of formal languages, for which we refer to [8, 9, 7].

This paper is organized as follows. We begin with the problem definition in section 2. In section 3 we provide a simple intuitive solution to the problem. While its time complexity of $O(n|\Sigma|^2)$ will be subsequently improved in section 4, it still provides an easy understanding of our solution. Section 4 improves the time of our algorithm to $O(n|\Sigma| \log n \log |\Sigma|)$. This improvement is possible by using a *naming* technique on the fingerprints. Section 5 describes the implementation of the renaming. We conclude with a discussion on how to answer various queries on fingerprint statistics (section 6) and open problems (section 7).

## 2 Problem Definition

**Definition 1** *Let $S = s_1 s_2 \cdots s_n$ be a string over a finite, ordered alphabet $\Sigma$. Let $S' = s_i s_{i+1} \cdots s_j$ be a substring of $S$ of length $j - i + 1$. The* fingerprint *of $S'$ is the ordered subset $\Sigma' \subseteq \Sigma$ of symbols appearing in $S'$.*

Formally we give an algorithm for the following problem:

**The Fingerprint Computation Problem**

*INPUT:* String $S = s_1 \cdots s_n$ over finite, ordered alphabet $\Sigma$.

*OUTPUT:* The number of distinct fingerprints of all the substrings $s_i \cdots s_j$, $\forall i, j,\ 1 \leq i \leq j \leq n$. This number is $< n|\Sigma|$.

**Example 1** *The number of distinct fingerprints of the substrings of the string $S = dccbcbabbbc$ is 10 - $(a); (b); (c); (d);\ (c,d); (b,c); (a,b);\ (b,c,d); (a,b,c); (a,b,c,d)$.*

For ease of exposition, throughout this paper we assume $\Sigma = \{1, \ldots, |\Sigma|\}$. This enables us to treat a symbol as an index.

This assumption can be made without loss of generality, since a $O(n \log n)$ preprocessing of the text is sufficient to construct an equivalent text over alphabet $\{1, ..., |\Sigma|\}$. Translating query $\phi$ to an equivalent query in the new alphabet can also be done in time $O(|\phi| \log n)$. Both the added preprocessing time and query translation time are subsumed by the time of the algorithms we present for texts over alphabet $\{1, ..., |\Sigma|\}$.

## 3 An $O(n|\Sigma|^2)$ Algorithm

We start with an $O(n|\Sigma|^2)$ algorithm that finds all distinct fingerprints of size $k$, for every $k$ with $1 \leq k \leq |\Sigma|$, of the substrings of $S$. The intuition behind our idea is similar to the linear algorithm for computing the sum of every consecutive $k$ elements of an array. The idea there is to move a window of size $k$ along the array adding the rightmost element and subtracting the leftmost element.

In our application this window has variable size up to $n$ since it must contain exactly $k$ *distinct* elements, each of which may occur more than once.

**Definition 2** *A* (variable length) window *is a pair* $(i, j)$, $1 \leq i \leq j \leq n$. *The substring* $s_i s_{i+1} \cdots s_j$ *is the substring* within *the window* $(i, j)$. *A variable length window* defines a $k$ sized fingerprint *if there are* exactly $k$ *different alphabet symbols within that window.*

**Algorithm's Idea:**

Let window $(i, j)$ define a $k$ sized fingerprint. We may move the right boundary of the window to the right (increase $j$) as long as no new symbol is encountered, and the new window still defines the same $k$ sized fingerprint. Once a new symbol is encountered, we move the left boundary to the right (increase $i$) until one symbol is dropped and we have a new window which defines a $k$ sized fingerprint.

We use the *trie* [10] data structure to compare fingerprints.

**Definition 3** *A* trie *$T$ for a set of strings $\{S_1, \cdots, S_r\}$ is a rooted directed tree satisfying:*

1. *Each edge is labeled with a character, and no two edges emanating from the same node have the same label.*
2. *Each node $v$ is associated with a string, denoted by $L(v)$, obtained by concatenating the labels on the path from the root to $v$, $L(root)$ is the empty string.*
3. *There is a node $v$ in $T$ if and only if $L(v)$ is a prefix of some string $S_j$ in the set.*

**Algorithm's Implementation:**

The algorithm maintains the following data structures:

- Two pointers $i_{\mathsf{left}}$ and $i_{\mathsf{right}}$. At every iteration $(i_{\mathsf{left}}, i_{\mathsf{right}})$ is the window under consideration.
- An array $COUNTER[1..|\Sigma|]$, where $COUNTER[j]$ is the number of occurrences of letter $j$ in the string $s_{i_{\mathsf{left}}} \cdots s_{i_{\mathsf{right}}}$.
- A binary array $LIFE[1..|\Sigma|]$ that represents the letters of the fingerprint of $s_{i_{\mathsf{left}}} \cdots s_{i_{\mathsf{right}}}$:

$$LIFE[j] = \begin{cases} 0, & \text{if } COUNTER[j] = 0; \\ 1, & \text{otherwise.} \end{cases}$$

The $LIFE$ array is an implementation device to easily allow representing the fingerprint. It is a "bit-vector" of all alphabet symbols, with the symbols in the fingerprint set to 1, and the other symbols set to 0. Other schemes to represent the fingerprint could have been used. We choose using the $LIFE$ array for its simplicity, and because we will use it in more sophisticated schemes in later sections.

- A variable *number* that counts the number of distinct letters in $s_{i_{\mathsf{left}}} \cdots s_{i_{\mathsf{right}}}$.
- A trie of all fingerprints of size $k$ in $S$.

We are now ready for the algorithm:

In the initialization stage we construct $(i_{\mathsf{left}}, i_{\mathsf{right}})$, the smallest leftmost window that defines a $k$ sized fingerprint. At each step $COUNTER[s_{i_{\mathsf{right}}}]$ is incremented by one. When it is changed from 0, *number* is incremented by one, and $LIFE[s_{i_{\mathsf{right}}}]$ is changed to 1. The move stops when $number = k$, and then the fingerprint is added to the trie.

**Initialization**

$i_{\mathsf{left}} \leftarrow 1$
$COUNTER, LIFE, number, i_{\mathsf{right}} \leftarrow 0$

Repeat until $number = k$:
  $i_{\mathsf{right}} \leftarrow i_{\mathsf{right}} + 1$
  $COUNTER[s_{i_{\mathsf{right}}}] \leftarrow COUNTER[s_{i_{\mathsf{right}}}] + 1$
  if $COUNTER[s_{i_{\mathsf{right}}}] = 1$ then $number \leftarrow number + 1$
    $LIFE[s_{i_{\mathsf{right}}}] \leftarrow 1$

{ Subroutine $HandleFingerprint$ adds the $k$ sized fingerprint defined by $(i_{\mathsf{left}}, i_{\mathsf{right}})$ to the fingerprint trie. }
Call $HandleFingerprint$

**end Initialization**

The main part of the algorithm consists of pairs of moves. In each one $i_{\mathsf{right}}$ is moved to the right until *number* exceeds $k$, then $i_{\mathsf{left}}$ is moved to the right until *number* goes down to $k$. At that point a new $k$ sized fingerprint is achieved and should be updated in the trie.

A move of $i_{\mathsf{right}}$: At each step $COUNTER[s_{i_{\mathsf{right}}}]$ is incremented by one. If it is changed from 0, *number* is incremented by one, $LIFE[s_{i_{\mathsf{right}}}]$ is changed to 1, and the move ends.

A move of $i_{\mathsf{left}}$: At each step $COUNTER[s_{i_{\mathsf{left}}}]$ is decremented by one. If it is changed to 0, *number* is decremented by one, $LIFE[s_{i_{\mathsf{left}}}]$ is changed to 0, and the move ends.

**Main Part of Algorithm**

Repeat until $i_{\text{right}} = n$

$\{i_{\text{right}}$ Move $\}$
Repeat until $number = k + 1$ or $i_{\text{right}} = n$
$i_{\text{right}} \leftarrow i_{\text{right}} + 1$
$COUNTER[s_{i_{\text{right}}}] \leftarrow COUNTER[s_{i_{\text{right}}}] + 1$
if $COUNTER[s_{i_{\text{right}}}] = 1$ then $number \leftarrow number + 1$
$LIFE[s_{i_{\text{right}}}] \leftarrow 1$

$\{i_{\text{left}}$ Move $\}$
If $i_{\text{right}} = n$ and $number \leq k$ then **end**
Repeat until $number = k$
$COUNTER[s_{i_{\text{left}}}] \leftarrow COUNTER[s_{i_{\text{left}}}] - 1$
if $COUNTER[s_{i_{\text{left}}}] = 0$ then $number \leftarrow number - 1$
$LIFE[s_{i_{\text{left}}}] \leftarrow 0$
$i_{\text{left}} \leftarrow i_{\text{left}} + 1$

Call $HandleFingerprint$

**end Main Part of Algorithm**

Below we present a straightforward but inefficient implementation of $HandleFingerprint$. This is done using the $LIFE$ array. $LIFE$ represents the fingerprint in a manner that does not depend on the order of the letters in the string but only on their lexicographical order. The trie may hold extra information depending on the queries we are to answer. For example, to be able to answer how many different fingerprints exist in $S$, all that is necessary is to count the number of leaves in the trie. To answer how many substrings have a given fingerprint, we may add to every leaf in the trie a counter that is incremented every time a fingerprint is found.

**Subroutine** $HandleFingerprint$

$fingerprint \leftarrow \lambda$ $\{\lambda$ represents the null string. $\}$
For $i = 1$ to $|\Sigma|$ do:
if $LIFE[i] = 1$ then concatenate Symbol $i$ to the right of string $fingerprint$

Add string $fingerprint$ to trie, with its leaf's counter set to 1. If it is already there then increment its leaf's counter by 1.

**end Subroutine**

**Time:** At every iteration, either $i_{\text{right}}$ or $i_{\text{left}}$ is incremented, thus, for a given $k$, moving on $S$ takes $O(n)$ time. In the current implementation, adding a fingerprint to the trie takes $O(|\Sigma|)$ time. For a fixed $k$ there are $O(n)$ calls to $HandleFingerprint$. Thus for a fixed $k$ the running time is $O(n|\Sigma|)$. $k$ ranges from 1 to $|\Sigma|$; hence, the total running time is $O(n|\Sigma|^2)$.

## 4 An $O(n|\Sigma|\log(n)\log(|\Sigma|))$ Algorithm

In this section we present a different idea for the bookkeeping of the fingerprints. We present a new subroutine *HandleFingerprint*. The other parts of the algorithm remain unchanged.

Instead of keeping the fingerprints in a trie, each distinct fingerprint is given a unique name. The names are given by using the *naming* technique [1, 4], which is a modified version of the algorithm of Karp, Miller and Rosenberg [3].

**The Naming technique:** Let $A$ be an array of size $m$. Assume, for the sake of simplicity, that $m$ is a power of 2, i.e. there is some $b$ such that $m = 2^b$. (If $m$ is not a power of 2, $A$ can be extended to an appropriate size by concatenating to its end a substring of a repeated single symbol. The size of the resulting string is no more than twice the size of the original string.)

A name is given to each subarray of size $2^i$ that starts on a position $\ell 2^i + 1$ in the array, where $0 \le i \le b$ and $0 \le \ell < m/2^i$. Names are given first to subarrays of size 1 then $2, 4, \ldots, 2^{b-1}$, at the end a name is given to the entire array.

A subarray of size $2^i$ is a concatenation of 2 subarrays of size $2^{i-1}$. The names of these 2 subarrays are used as the input for the computation of the name of the subarray of size $2^i$. The process may be viewed as constructing a complete binary tree, which we will refer to as a *naming tree*. The leaves of the tree (level 0) are the elements of the initial array. Node $x$ in level $i$ is the parent of nodes $2x-1$ and $2x$ in level $i-1$. See example 2. Note that for an array of length $m$, at most $2m-1$ names are given. Our implementation of the naming technique is shown in Section 5.

**Example 2** *Below is the result of naming string* 0110001010110010*:*

<table>
<tr><td colspan="16">11</td></tr>
<tr><td colspan="8">9</td><td colspan="8">10</td></tr>
<tr><td colspan="4">6</td><td colspan="4">7</td><td colspan="4">8</td><td colspan="4">7</td></tr>
<tr><td colspan="2">2</td><td colspan="2">3</td><td colspan="2">4</td><td colspan="2">3</td><td colspan="2">3</td><td colspan="2">5</td><td colspan="2">4</td><td colspan="2">3</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
</table>

We will use naming for handling the fingerprints. However, we do not use naming on the fingerprint itself, since the changes from fingerprint to fingerprint require too much effort. Rather, we use naming on the $LIFE$ array. As previously mentioned, an instance of $LIFE$ represents a fingerprint. During one successful move of the variable length window, $LIFE$ changes *exactly twice*, one bit is added (the new alphabet symbol) and one bit gets deleted (the deleted symbol).

**Example 3** *Assume that the string* 0110001010110010 *from example 2 represents an instance of the $LIFE$ array. Suppose the window move adds the* 10*th alphabet symbol, i.e. the $LIFE$ array changes to* 0110001011110010*. In the diagram below we indicate in boldface the names that changed as a result of the change to the string.*

<table>
<tr><td colspan="16" align="center">14</td></tr>
<tr><td colspan="8" align="center">9</td><td colspan="8" align="center">13</td></tr>
<tr><td colspan="4" align="center">6</td><td colspan="4" align="center">7</td><td colspan="4" align="center">12</td><td colspan="4" align="center">7</td></tr>
<tr><td colspan="2" align="center">2</td><td colspan="2" align="center">3</td><td colspan="2" align="center">4</td><td colspan="2" align="center">3</td><td colspan="2" align="center">5</td><td colspan="2" align="center">5</td><td colspan="2" align="center">4</td><td colspan="2" align="center">3</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
</table>

From example 3 one can see that a single change in an array of size $m$ causes at most $\log m$ names to change, since there is at most one name change in every level. Formally:

**Observation 1** *Let $A$ be an array of length $m$ and let $B$ be an array of length $m$ derived by changing the value of a single element of $A$. Then for every level in the naming tree, there is a single name that requires a change. Since there are $\log m$ levels, then only $\log m$ names need to be changed in order to compute the name of $B$.*

We conclude from observation 1 that at every change of the variable length window, only $O(\log|\Sigma|)$ names need to be handled, since only two locations of $LIFE$ are changed.

The subroutine $HandleFingerprint$ will now look as follows:

**Subroutine** $HandleFingerprint$ (High Level)

Compute name $life$ of array $LIFE$
If $life$ is a new name, then set its counter to 1
If $life$ appeared previously, then increment its counter by 1

**end Subroutine**

**Time:** In section 5 we show an implementation of $HandleFingerprint$ in time $O(\log n \log|\Sigma|)$. This means that for a fixed fingerprint size our algorithm's running time is $O(n \log n \log|\Sigma|)$. $k$ ranges from 1 to $|\Sigma|$; hence, the total running time is $O(n|\Sigma| \log(n) \log(|\Sigma|))$.

# 5 Computing Names

We have seen in section 4 that the name of the $LIFE$ array can be maintained at a cost of $O(\log|\Sigma|)$ per change, which is the number of queries to the name data base. Subroutine *HandleFingerprint* requires the knowledge of whether the updated $LIFE$ array gets a new name, or a name that appeared previously. Before we show an efficient implementation of this task, let us bound the maximum number of different names our algorithm needs to generate for a fixed fingerprint size $k$.

**Lemma 1** *The maximum number of different names generated by our algorithm's naming of size $k$ fingerprints on a text of length $n$ is $O(n \log|\Sigma|)$. The maximum number of names generated at a fixed level $i$ in the naming tree is $O(n)$.*

**Proof**: The first fingerprint initializes the $LIFE$ array. Naming the initial $LIFE$ array requires at most $2|\Sigma|-1$ names ($O(n)$). Throughout the algorithm, at most $n$ changes to the initial fingerprint are possible. Assume, in the worst case, that every change creates a new fingerprint. Observation 1 guarantees that for every change in the fingerprint, no more than one change occurs in every level of the naming tree. Therefore, the maximum number of different possible names at every level is $2n$. Since there are $O(\log|\Sigma|)$ levels in the tree, then the maximum possible number of different names necessary for a fixed fingerprint size $k$ is $O(n \log|\Sigma|)$. ∎

Our naming strategy is as follows. A name is a pair of previous names. At level $i$ of the naming, we compute the name of subarray $A_1 A_2$ of size $2^i$, where $A_1$ and $A_2$ are consecutive subarrays of size $2^{i-1}$ each. We give as names the natural numbers in increasing order. Notice that every level only uses the names of the level below it, thus the names we use at every level are numbers from the set $\{1, ..., n\}$.

To give an array a name, we need only to know if the pair of names of the composing subarrays has appeared previously. If it did, then the array gets the name of this pair. Otherwise, it gets a new name. It is necessary, therefore, to show a quick way to dynamically access pairs of numbers from a bounded range universe. Formally, we would like a solution to the following problem:

**Definition 4** The dynamic pair recognition problem *is the following:*
*INPUT: A sequence of queries* $\{(a_j, b_j)\}_{j=1}^{\infty}$, *where* $a_j, b_j \in \{1, ..., j\}$.
*OUTPUT: Dynamically decide, for every query* $(a_j, b_j)$, *whether there exist* $c$, $c < i$ *such that* $(a_j, b_j) = (a_c, b_c)$.

We will present a solution that requires, for solving each query $(a_j, b_j)$, time $O(\log x)$, where $x$ is the number of previous queries whose first pair element is $a_j$. In our case, since in every level there are at most $O(n)$ different numbers, a dynamic pair recognition query is solved in time $O(\log n)$. A dynamic pair recognition query is asked $O(\log|\Sigma|)$ times for each fingerprint. We conclude:

**Claim 1** *The running time of HandleFingerprint is* $O(\log|\Sigma| \log n)$.

In the remainder of this section we present the solution to the dynamic pair recognition problem. Note that our pair recognition problem is not really dynamic, since all pairs of level $i-1$ are available before processing of level $i$ begins. Thus it is possible to construct an $n \times n$ matrix initialized as 0, and fill in all pairs as they are encountered. This allows solving each pair query in constant time but the initial cost is $O(n^2)$. We presented the problem as a dynamic problem. While every query will take time $O(\log n)$, there are only a total of $O(n)$ queries, so our total time is $O(n \log n)$, which is faster.

**Intuition:**

At any point $j$ the pairs we are considering all have their first element no greater than $j$. Thus, accessing the first element can be done in constant time by direct access. This suggest "gathering" all pairs in trees rooted at their first element. However, if we make sure these trees are ordered by the second element and balanced, we can find elements by binary search in time that is logarithmic in the tree size.

**Algorithm's Implementation:**

The algorithm maintains the following data structure:

- $BAL[a]$ is a balanced binary tree of all pairs $(a, b)$ that have been named so far, sorted by $b$. Since $a, b$ are increasing natural numbers, starting from 1, $BAL[a]$ is directly accessed by $a$

The algorithm is now straightforward. We are given pair $(a, b)$ at time $j$ and need to recognize if it has appeared so far.

**Pair Recognition Algorithm**

if $(a, b) \in BAL[a]$ then output "occurred previously, name is name$(a, b)$"
    else:
        $j \leftarrow j + 1$
        add $(a, b)$ to $BAL[a]$
        name$(a, b) \leftarrow j$
        initialize empty $BAL[j]$

**end Algorithm**

**Time:** It is clear that the time for the pair recognition algorithm is the time for searching the balanced tree, i.e. $O(\log |BAL[a]|) = O(\log(n))$.

## 6 Fingerprint Statistics

The above algorithm allows us to efficiently name every fingerprint encountered. This scheme easily allows answering a number of queries on fingerprint statistics.

**Query 1**
*INPUT:* $k$.
*OUTPUT:* The number of different size $k$ fingerprints in $S$.

The answer to the above query can be provided immediately if one keeps count of the number of top level names for every $k$.

Another type of query that interests us is providing the number of substrings of $S$ that have a given fingerprint. This query requires some discussion. Consider the string *xabcabcabcy* and suppose $k = 3$. Clearly the window $(2, 10)$ defines a substring whose fingerprint is *abc*. However, so does every substring of *abcabcabc* (that is, every subwindow of $(2, 10)$) whose length is at least 3. When we want to count the number of substrings whose fingerprint is *abc*, which number do we count? This brings us to a sharpening of the definition. (We assume $s_0$ and $s_{n+1}$ are defined and do not belong to $\Sigma$.)

**Definition 5** *Let $S = s_1 s_2 \cdots s_n$ be a string over finite, ordered alphabet $\Sigma$. Let $\phi$ be a fingerprint. We call substring $s_i \cdots s_j$* $\phi$-maximal *if $s_{i-1}$ and $s_{j+1}$ do not belong to fingerprint $\phi$. We say that $s_i \cdots s_j$ is* $\phi$-minimal *if both $s_i$ and $s_j$ appear only once in the substring (i.e. removal of either of them will change the fingerprint).*

*If $s_i \cdots s_j$ is $\phi$-maximal ($\phi$-minimal) and $\phi$ has size $k$ then we say that $s_i \cdots s_j$ is* $k$-maximal *(*$k$-minimal*).*

We can now formally phrase Query 2:

**Query 2**
*INPUT:* Fingerprint $\phi$.
*OUTPUT:* The number of $\phi$-maximal ($\phi$-minimal) substrings in $S$.

Maximal substrings:

The movements of the $i_{\text{right}}$ and $i_{\text{left}}$ pointers in the main algorithm of section 3 define the maximality of the substring. If we let $i_{\text{right}}$ continue *as long as number* $= k+1$ (rather than *until number* $= k+1$) and we leave the advance of $i_{\text{left}}$ as originally written, then we will get windows that provide maximal substrings.

Minimal substrings:

**Lemma 2** *For $k \geq 3$, substring $s_i \cdots s_j$ is a $k$-minimal substring iff $s_{i+1} \cdots s_{j-1}$ is a $(k-2)$-maximal substring and $s_i \neq s_j$.*

**Proof:** Assume $s_i \cdots s_j$ is $k$-minimal. Then clearly $s_i \neq s_j$, otherwise one of them could be dropped without changing the fingerprint which would contradict the substring's $k$-minimality. For the same reason $s_i$ does not appear in $s_{i+1} \cdots s_j$ and $s_j$ does not appear in $s_i \cdots s_{j-1}$. This means that $s_{i+1} \cdots s_{j-1}$ has fingerprint $k-2$. Extending the substring on any side raises its fingerprint size. Therefore by definition 5 $s_{i+1} \cdots s_{j-1}$ is $(k-2)$ maximal.

Conversely, if $s_{i+1} \cdots s_{j-1}$ is $(k-2)$ maximal then by definition 5 $s_i$ does not appear in $s_{i+1} \cdots s_{j-1}$ and $s_j$ does not appear in $s_{i+1} \cdots s_{j-1}$. If, in addition, $s_i \neq s_j$ then the fingerprint of $s_i \cdots s_j$ is of size $k$, and is minimal since dropping $s_i$ or $s_j$ reduces the fingerprint size by 1. ∎

Using Lemma 2, to find $k$-minimal substrings we simply check, for any $(k-2)$-maximal substring, if the letter preceding the substring is not equal to the letter following it. Hence, the main algorithm gives the mechanism to count maximal or minimal substrings. Note that the $(k-2)$-maximal substrings and the $k$-minimal substrings are computed together.

We are now ready to tackle Query 2. At the time of fingerprint naming maintain, for every name (representing a maximal or minimal fingerprint), a counter for the number of times it was encountered. Subsequently, the answer to the above query can be provided by computing the name of fingerprint $\phi$. The time for mapping $\phi$ to its $LIFE$ bit notation is $O(|\Sigma|)$. The name is then computed in time $O(|\Sigma| \log n)$. The number of times it appears is denoted by the name.

A similar problem that could be of interest is finding *all* substrings with a given fingerprint $\phi$. The total number of substrings with fingerprint $\phi$ can be easily computed using the following immediate observations.

**Observation 2**

1. *No two different $\phi$-maximal substrings overlap.*
2. *If $S_1$ is a $\phi$-maximal substring and $S_2$ is a $\phi$-minimal substring then either $S_2$ is a substring of $S_1$ or $S_1$ and $S_2$ don't overlap.*

3. *Every substring with fingerprint $\phi$ is contained in a $\phi$-maximal substring and contains at least one $\phi$-minimal substring.*

For every pair of substrings $\langle S_1, S_2 \rangle$ where $S_1$ is $\phi$-maximal, $S_2$ is $\phi$-minimal, and $S_2$ is a substring of $S_1$ (let $S_1 = s_i \cdots s_j$ and $S_2 = s_{i+n_\ell} \cdots s_{j-n_r}$) the pair $\langle S_1, S_2 \rangle$ contributes $(n_\ell + 1)(n_r + 1)$ substrings with fingerprint $\phi$.

## 7 Open Problems

Recall that we can only solve Query 2 (for input fingerprint $\phi$ count the number of substring occurrences whose fingerprint is $\phi$) by mapping $\phi$ to its $LIFE$ notation. This automatically lower bounds the time by $O(|\Sigma|)$, even for small $|\phi|$. Is it possible to answer Query 2 in time $O(|\phi| \times \text{polylog } n)$?

It may be possible to improve the algorithm by a constant factor if the position of $i_{\mathsf{left}}$ only moves after computing fingerprints of *all* sizes starting at location $i_{\mathsf{left}}$, rather than the proposed method of computing the fingerprints size by size. However, the algorithm will still have $|\Sigma|$ as a multiplicative factor. It would be interesting to see if the $|\Sigma|$ factor can also be reduced. If such a reduction is possible, it will probably involve a different idea, perhaps one that computes the fingerprints without recourse to the greater amount of information provided by the Parikh vector. Is such a method possible?

Finally, Parikh vectors *per se* and their natural generalizations to weighted alphabets find possible use in a number of applications, e.g., approximate string searching in biosequences and other textfiles in which the individual symbols carry some weight. Some problems thus revolve around the existence of non-trivial extensions to these formulations of the techniques developed in this paper.

## References


[1] A. Apostolico, C. Iliopoulos, G.M. Landau, B. Schieber, and U. Vishkin. Parallel construction of a suffix tree with applications. *Algorithmica*, 3:347–365, 1988.

[2] F. Karlsson, A. Voutilainen, J. Heikkilä, and A. Anttila. *Constraint Grammar. A Language Independent System for Parsing Unrestricted Text.* Mouton de Gruyter, 1995.

[3] R. Karp, R. Miller, and A. Rosenberg. Rapid identification of repeated patterns in strings, arrays and trees. *Symposium on the Theory of Computing*, 4:125–136, 1972.

[4] Z. M. Kedem, G. M. Landau, and K. V. Palem. Parallel suffix-prefix matching algorithm and application. *SIAM J. Comp*, 25(5):998–1023, 1996.

[5] P. Langley. *Elements of Machine Learning.* Morgan Kaufmann, San Francisco, 1995.

[6] T. M. Mitchell. *Machine Learning.* McGraw-Hill, New York, 1997.

[7] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice Hall, Englewood Cliffs, New Jersey, 1982.

[8] R. J. Parikh. On context-free languages. *Journal of the ACM*, 14(4):570–581, 1966.

[9] A. Salomaa. *Formal Languages.* Academic Press, New York and London, 1973.

[10] R. Sedgewick. *Algorithms.* Addison-Wesley, second edition, 1988.