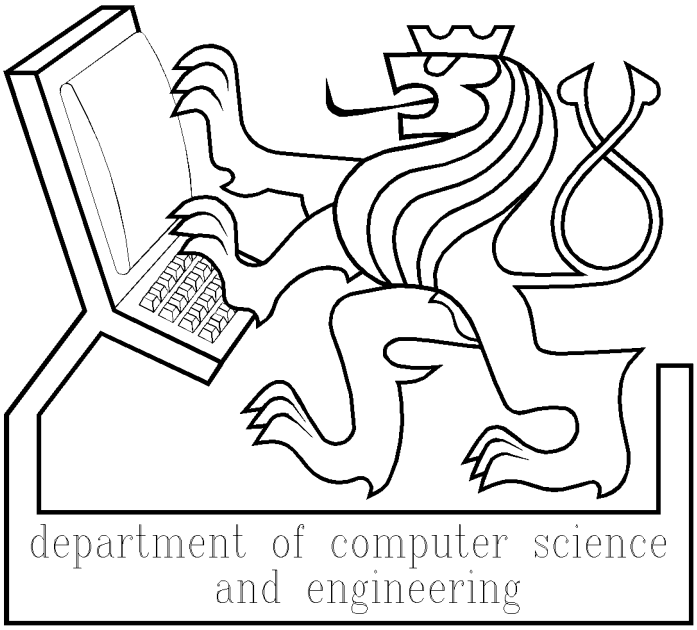
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Proceedings of the Prague Stringology Conference '05

Edited by Jan Holub and Milan Šimánek



Czech Technical University
Prague
Czech Republic

Proceedings of the Prague Stringology Conference '05

Edited by Jan Holub and Milan Šimánek

August 2005

Department of Computer Science and Engineering
Faculty of Electrical Engineering
Czech Technical University in Prague
Karlovo nám. 13
121 35 Prague 2
Czech Republic

Program Committee

Amihood Amir, Gabriela Andrejková, Jun-ichi Aoe, Maxime Crochemore, František Franěk, Jan Holub, Costas S. Iliopoulos, Shmuel Klein, Thierry Lecroq, Bořivoj Melichar (chair), Yoan J. Pinzon, Marie-France Sagot, Bruce W. Watson

Organizing Committee

Miroslav Balík, Jan Holub, Bořivoj Melichar, Milan Šimánek

URL

`http://cs.felk.cvut.cz/psc`

Proceedings of the Prague Stringology Conference '05
Edited by Jan Holub and Milan Šimánek
Published by: Prague Stringology Club
Department of Computer Science and Engineering
Faculty of Electrical Engineering
Czech Technical University in Prague
Karlovo nám. 13, Praha 2, 121 35, Czech Republic.
E-mail: psc@cs.felk.cvut.cz Phone: +420-2-2435-7470
Printed by Vydavatelství ČVUT, Zikova 4, Praha 6, 166 35, Czech Republic

ISBN 80-01-03307-4

Table of Contents

Preface

The Prague Stringology Conference 2005 (PSC'05) was held at the Department of Computer Science and Engineering of the Czech Technical University in Prague, Czech Republic, on August 29–31, 2005. The conference focused on stringology and related topics. Stringology is a discipline concerned with algorithmic processing of strings and sequences.

The papers submitted were reviewed by the program committee and sixteen were selected for presentation at the conference, based on originality and quality. This volume contains not only these selected papers but also two invited talks devoted to taxonomy of suffix array construction algorithms and asynchronous pattern matching.

PSC'05 is the tenth event of the Prague Stringology Club. In the years 1996–2000 the Prague Stringology Club Workshops (PSCW's) and the Prague Stringology Conferences (PSC's) in 2001–2004 preceded this conference. The proceedings of these workshops and the conferences had been published by Czech Technical University in Prague and are available on WWW pages of the Prague Stringology Club. Selected contributions were published in special issues of the journal Kybernetika, the Nordic Journal of Computing, the Journal of Automata, Languages and Combinatorics, and the International Journal of Foundations of Computer Science.

The Prague Stringology Club was founded in 1996 as a research group at the Department of Computer Science and Engineering of the Czech Technical University in Prague. The goal of the Prague Stringology Club is to study algorithms on strings and sequences with emphasis on finite automata theory. The first event organized by the Prague Stringology Club was the workshop PSCW'96 featuring only a handful invited talks. However, since PSCW'97 the papers and talks are selected by a rigorous peer review process. The objective is not only to present new results in stringology, but also to facilitate personal contacts among the people working on these problems.

I would like to thank all those who had submitted papers for PSC'05 as well as the reviewers. Special thanks goes to all the members of the program committee, without whose efforts it would not have been possible to put together such a stimulating program of PSC'05. Last, but not least, my thanks go to the members of the organizing committee for ensuring such a smooth running of the conference.

In Prague, Czech Republic
on August 2005
Jan Holub

A Taxonomy of Suffix Array Construction Algorithms*

Simon J. Puglisi[1], W. F. Smyth[1,2], and Andrew Turpin[3]

[1] Department of Computing, Curtin University, GPO Box U1987
Perth WA 6845, Australia
e-mail: puglissj@computing.edu.au

[2] Algorithms Research Group, Department of Computing & Software
McMaster University, Hamilton ON L8S 4K1, Canada
e-mail: smyth@mcmaster.ca
www.cas.mcmaster.ca/cas/research/groups.shtml

[3] School of Computer Science & Information Technology
RMIT University, GPO Box 2476V
Melbourne V 3001, Australia
e-mail: aht@cs.rmit.edu.au

Abstract. In 1990 Manber & Myers proposed suffix arrays as a space-saving alternative to suffix trees and described the first algorithms for suffix array construction and use. Since that time, and especially in the last few years, suffix array construction algorithms have proliferated in bewildering abundance. This survey paper attempts to provide simple high-level descriptions of these numerous algorithms that highlight both their distinctive features and their commonalities, while avoiding as much as possible the complexities of implementation details. We also provide comparisons of the algorithms' worst-case time complexity and use of additional space, together with results of recent experimental test runs on many of their implementations.

1 Introduction

Suffix arrays were introduced in 1990 by Manber & Myers [MM90, MM93], along with algorithms for their construction and use as a space-saving alternative to suffix trees. In the intervening fifteen years there have certainly been hundreds of research articles published on the construction and use of suffix trees and their variants. Over that period, it has been shown that

- practical space-efficient suffix array construction algorithms (SACAs) exist that require worst-case time linear in string length [KA03, KS03];
- SACAs exist that are even faster in practice, though with supralinear worst-case construction time requirements [LS99, BK03, MF04, M05];

*Supported in part by grants from the Natural Sciences & Engineering Research Council of Canada and the Australian Research Council.

- any problem whose solution can be computed using suffix trees is solvable with the same asymptotic complexity using suffix arrays [AKO04].

Thus suffix arrays have become the data structure of choice for many, if not all, of the string processing problems to which suffix tree methodology is applicable.

In this survey paper we do not attempt to cover the entire suffix array literature. Our more modest goal is to provide an overview of SACAs, in particular those modeled on the efficient use of main memory — we exclude the substantial literature (for example, [CF02]) that discusses strategies based on the use of secondary storage. Further, we deal with the construction of compressed ("succinct") suffix arrays only insofar as they relate to standard SACAs. For example, algorithms such as those of Grossi et al. and references therein [GGV04] are not covered.

Section 2 provides an overview of the SACAs known to us, organized into a "taxonomy" based primarily on the methodology used. As with all classification schemes, there is room for argument: there are many cross-connections between algorithms that occur in disjoint subtrees of the taxonomy, just as there may be between species in a biological taxonomy. Our aim is to provide as comprehensive and, at the same time, as accessible a description of SACAs as we can.

Also in Section 2 we present the vocabulary to be used for the structured description of each of the algorithms that will be given in Section 3. Then in Section 4, we report on the results of experimental results on many of the algorithms described and so draw conclusions about their relative speed and space-efficiency.

2 Overview

We consider throughout a finite nonempty ***string*** $\boldsymbol{x} = \boldsymbol{x}[1..n]$ of ***length*** $n \geq 1$, defined on an ***indexed*** alphabet Σ; that is,

- the letters $\lambda_j, j = 1, 2, \ldots, \sigma$ of $|\Sigma|$ are ordered: $\lambda_1 < \lambda_2 < \cdots < \lambda_\sigma$;
- an array $A[\lambda_1..\lambda_\sigma]$ can be defined in which, for every $j \in 1..\sigma$, $A[\lambda_j]$ is accessible in constant time;
- $\lambda_\sigma - \lambda_1 \in O(n)$.

Essentially, we assume that Σ can be treated as a sequence of integers whose range is not too large. Typically, the λ_j may be represented by ASCII codes 0..255 (English alphabet) or binary integers 00..11 (DNA) or simply bits, as the case may be. We shall generally assume that a letter can be stored in a byte and that n can be stored in one computer word (four bytes).

The use of terminology not defined here follows [S03].

We are interested in computing the ***suffix array*** of $\boldsymbol{x}$, which we write $\mathrm{SA}_{\boldsymbol{x}}$ or just SA; that is, an array SA$[1..n]$ in which SA$[j] = i$ iff $\boldsymbol{x}[i..n]$ is the j^{th} suffix of $\boldsymbol{x}$ in (ascending) lexicographical order (***lexorder***). For simplicity we will frequently refer to $\boldsymbol{x}[i..n]$ simply as "suffix i"; also, it will often be convenient for processing to incorporate into $\boldsymbol{x}$ at position n an ending sentinel \$ assumed to be less than any λ_j.

Then, for example, on alphabet $\Sigma = \{\$, a, b, c, d, e\}$:

	1	2	3	4	5	6	7	8	9	10	11	12
$\boldsymbol{x} =$	a	b	e	a	c	a	d	a	b	e	a	\$
SA =	12	11	8	1	4	6	9	2	5	7	10	3

Thus SA tells us that $\boldsymbol{x}[12..12] =$ \$ is the least suffix, $\boldsymbol{x}[11..12] = a$\$ the second least, and so on (alphabetical ordering of the letters assumed). Note that SA is always a permutation of $1..n$.

Often used in conjunction with $\mathrm{SA}_{\boldsymbol{x}}$ is the ***lcp array*** lcp$[1..n]$: for every $j \in 2..n$, lcp$[j]$ is just the ***longest common prefix*** of suffixes SA$[j-1]$ and SA$[j]$. In our example:

	1	2	3	4	5	6	7	8	9	10	11	12
$\boldsymbol{x} =$	a	b	e	a	c	a	d	a	b	e	a	\$
SA =	12	11	8	1	4	6	9	2	5	7	10	3
lcp =	$-$	0	1	4	1	1	0	3	0	0	0	2

Thus the longest common prefix of suffixes 11 and 8 is 1, that of suffixes 8 and 1 is 4. Since lcp can be computed in linear time from $\mathrm{SA}_{\boldsymbol{x}}$ [KLAAP01, M04], also as a byproduct of some of the SACAs discussed below, we do not consider its construction further in this paper. However, the ***average lcp*** — that is, the average $\overline{\mathrm{lcp}}$ of the $n-1$ integers in the lcp array — is as we shall see a useful indicator of the relative efficiency of certain SACAs, notably Algorithm S.

We remark that both SA and lcp can be computed in linear time by a preorder traversal of a suffix tree.

Many of the SACAs also make use of the ***inverse suffix array***, written $\mathrm{ISA}_{\boldsymbol{x}}$ or ISA: an array ISA$[1..n]$ in which

$$\mathrm{ISA}[i] = j \iff \mathrm{SA}[j] = i.$$

ISA$[i] = j$ therefore says that suffix i has ***rank*** j in lexorder. Continuing our example:

	1	2	3	4	5	6	7	8	9	10	11	12
$\boldsymbol{x} =$	a	b	e	a	c	a	d	a	b	e	a	\$
ISA =	4	8	12	5	9	6	10	3	7	11	2	1

Thus ISA tells us that suffix 1 has rank 4 in lexorder, suffix 2 rank 8, and so on. Note that ISA is also a permutation of $1..n$, and so SA and ISA are computable, one from the other, in $\Theta(n)$ time:

for $j \leftarrow 1$ **to** n **do**
  $\mathrm{SA}\big[\mathrm{ISA}[j]\big] \leftarrow j$

As shown in Figure 1, this computation can if required also be done in place.

Many of the algorithms we shall be describing depend upon a partial sort of some or all of the suffixes of $\boldsymbol{x}$, partial because it is based on an ordering of the prefixes of these suffixes that are of length $h \geq 1$. We refer to this partial ordering as an ***h-ordering*** of suffixes into ***h-order***, and to the process itself as an ***h-sort***. If two or more suffixes are equal under h-order, we say that they have the same ***h-rank*** and therefore fall into the same ***h-group***; they are accordingly said to be ***h-equal***. Usually an h-sort is ***stable***, so that any previous ordering of the suffixes is retained within each h-group.

```
for j ← 1 to n do
  i ← SA[j]
 — Negative entries already processed
  if i > 0 then
    j0, j' ← j
    repeat
      temp ← SA[i]; SA[i] ← −j'
      j' ← i; i ← temp
    until i = j0
    SA[i] ← −j'
  else
    SA[j] ← −i
```

Figure 1: Algorithm for computing ISA from SA in place

The results of an h-sort are often stored in an approximate suffix array, written SA_h, and/or an approximate inverse suffix array, written ISA_h. Here is the result of a 1-sort on all the suffixes of our example string:

	1	2	3	4	5	6	7	8	9	10	11	12
$\boldsymbol{x} =$	a	b	e	a	c	a	d	a	b	e	a	\$
$\mathrm{SA}_1 = 12$	(1	4	6	8	11)	(2	9)	5	7	(3	10)	
$\mathrm{ISA}_1 = 2$	7	11	2	9	2	10	2	7	11	2	1	
or 6	8	12	6	9	6	10	6	8	12	6	1	
or 2	3	6	2	4	2	5	2	3	6	2	1	

The parentheses in SA_1 enclose 1-groups not yet reduced to a single entry, thus not yet in final sorted order. Note that SA_h retains the property of being a permutation of $1..n$, while ISA_h may not. Depending on the requirements of the particular algorithm, ISA_h may as shown express the h-rank of each h-group in various ways:

- the leftmost position j in SA_h of a member of the h-group, also called the ***head*** of the h-group;
- the rightmost position j in SA_h of a member of the h-group, also called the ***tail*** of the h-group;
- the ordinal left-to-right counter of the h-group in SA_h.

Compare the result of a 3-sort:

	1	2	3	4	5	6	7	8	9	10	11	12
$\boldsymbol{x} =$	a	b	e	a	c	a	d	a	b	e	a	\$
$\mathrm{SA}_3 = 12$	11	(1	8)	4	6	(2	9)	5	7	10	3	
$\mathrm{ISA}_3 = 3$	7	12	5	9	6	10	3	7	11	2	1	
or 4	8	12	5	9	6	10	4	8	11	2	1	
or 3	6	10	4	7	5	8	3	6	9	2	1	

Observe that an $(h{+}1)$-sort is a ***refinement*** of an h-sort: all members of an $(h{+}1)$-group belong to a single h-group.

We now have available a vocabulary sufficient to characterize the main species of SACA as follows.

(1) Prefix-Doubling

First a fast 1-sort is performed (since Σ is indexed, bucket sort can be used); this yields SA_1/ISA_1. Then for every $h = 1, 2, \ldots$, $\mathrm{SA}_{2h}/ISA_{2h}$ are computed in $\Theta(n)$ time from SA_h/ISA_h until every $2h$-group is a singleton. The time required is therefore $O(n \log n)$. There are two algorithms in this class: MM [MM90, MM93] and LS [S98, LS99].

(2) Recursive

Form strings $\boldsymbol{x'}$ and $\boldsymbol{y}$ from $\boldsymbol{x}$, then show that if $\mathrm{SA}_{\boldsymbol{x'}}$ is computed, therefore $\mathrm{SA}_{\boldsymbol{y}}$ and finally $\mathrm{SA}_{\boldsymbol{x}}$ can be computed in $O(n)$ time. Hence the problem of computing $\mathrm{SA}_{\boldsymbol{x'}}$ recursively replaces the computation of $\mathrm{SA}_{\boldsymbol{x}}$. Since $|\boldsymbol{x'}|$ is always chosen so as to be less than $2|\boldsymbol{x}|/3$, the overall time requirement of these algorithms is $\Theta(n)$. There are three main algorithms in this class: KA [KA03], KS [KS03] and KJP [KJP04].

(3) Induced Copying

The key insight here is the same as for the recursive algorithms — a complete sort of a selected subset of suffixes can be used to "induce" a complete sort of other subsets of suffixes. The approach however is nonrecursive: an efficient suffix sorting technique (for example, [BM93, MBM93, M97, BS97, SZ04]) is invoked for the selected subset of suffixes. The general idea seems to have been first proposed by Burrows & Wheeler [BW94], but it has been implemented in quite different ways [IT99, S00, MF04, SS05, BK03, M05]. In general, these methods are very efficient in practice, but may have worst-case asymptotic complexity as high as $O(n^2 \log n)$.

The goal is to design SACAs that

- have minimal asymptotic complexity $\Theta(n)$;
- are fast "in practice" (that is, on collections of large real-world data sets such as [H04]);
- are ***lightweight*** — that is, use a small amount of working storage in addition to the $5n$ bytes required by $\boldsymbol{x}$ and $\mathrm{SA}_{\boldsymbol{x}}$.

To date none of the SACAs that has been proposed achieves all of these objectives.

Figure 2 presents our taxonomy of the fourteen species of SACA that have been recognized so far; Table 1 summarizes their time and space requirements.

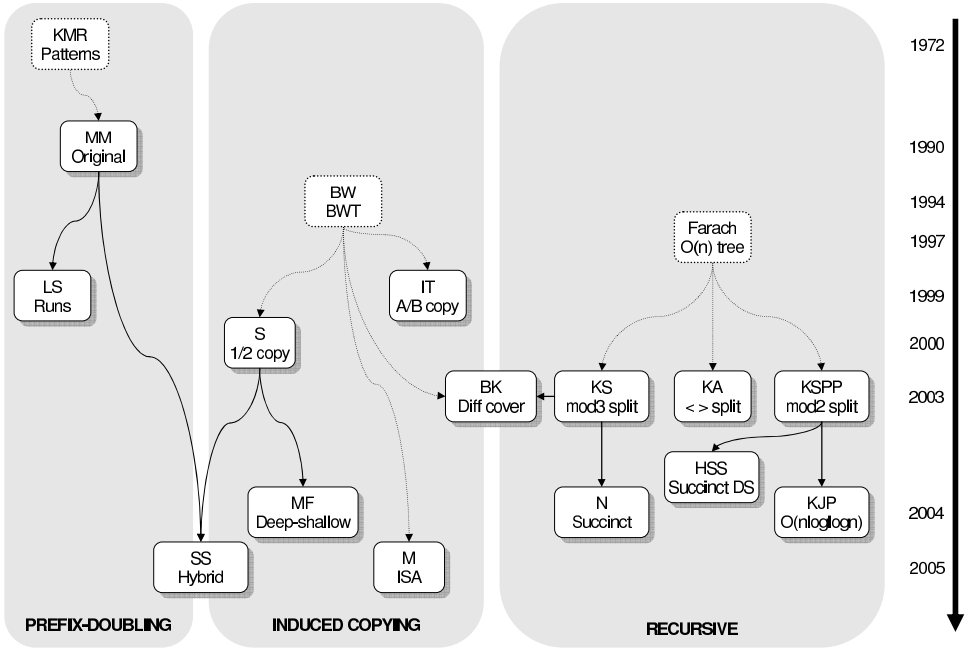


Figure 2: Taxonomy of suffix array construction algorithms

Table 1: Performance summary of the construction algorithms. Speed is relative to MF, the fastest in our experiments, and Memory is given in the number of bytes required including space required for the suffix array and input.

Algorithm	Worst Case	Speed	Memory
Prefix-Doubling			
MM [MM93]	$O(n \log n)$	16	$8n$
LS [LS99]	$O(n \log n)$	1.7	$8n$
Recursive			
KA [KA03]	$O(n)$	2.2	13-14n
KS [KS03]	$O(n)$	2.8	10-13n
KSPP [KSPP03]	$O(n)$	–	–
HSS [HSS03]	$O(n)$	–	–
KJP [KJP04]	$O(n \log\log n)$	2.1	13-16n
Induced Copying			
IT [IT99]	$O(n^2 \log n)$	4	$5n$
S [S00]	$O(n^2 \log n)$	2.1	$5n$
BK [BK03]	$O(n \log n)$	2.1	5-6n
MF [MF04]	$O(n^2 \log n)$	1	$5n$
SS [SS05]	$O(n^2)$	1	9-10n
M [M05]	$O(n^2 \log n)$	1	5-7n
Suffix Tree			
K [K99]	$O(n \log \sigma)$	4	15-20n

3 The Algorithms

3.1 Prefix-Doubling Algorithms [KMR72]

Here we consider algorithms that, given an h-order SA_h of the suffixes of $\boldsymbol{x}$, $h \geq 1$, compute a $2h$-order in $O(n)$ time. Thus prefix-doubling algorithms require at most $\log_2 n$ steps to complete the suffix sort and execute in $O(n \log n)$ time in the worst case.

Normally prefix-doubling algorithms initialize SA_1 for $h = 1$ using a linear-time bucket sort. The main idea [KMR72] is as follows:

Observation 1. *Suppose that* SA_h *and* ISA_h *have been computed for some* $h > 0$, *where* $i = \mathrm{SA}_h[j]$ *is the* j^{th} *suffix in* h*-order and* h-$\mathrm{rank}[i] = \mathrm{ISA}_h[i]$. *Then a sort using the integer pairs*

$$\big(\mathrm{ISA}_h[i], \mathrm{ISA}_h[i+h]\big)$$

as keys, $i+h \leq n$, *computes a* $2h$*-order of the suffixes* i. *(Suffixes* $i > n-h$ *are necessarily already fully ordered.)*

The two main prefix-doubling algorithms differ primarily in their application of this observation:

- Algorithm MM does an implicit $2h$-sort by performing a left-to-right scan of SA_h that induces the $2h$-rank of $\mathrm{SA}_h[j]-h$, $j = 1, 2, \ldots, n$;
- Algorithm LS explicitly sorts each h-group using the ternary-split quicksort (TSQS) of Bentley & McIlroy [BM93].

Manber & Myers [MM90, MM93]

Algorithm MM employs Observation 1 as follows:

> If SA_h is scanned left to right (thus in h-order of the suffixes), $j = 1, 2, \ldots, n$, then the suffixes
>
> $$i-h = \mathrm{SA}_h[j]-h > 0$$
>
> are necessarily scanned in $2h$-order within their respective h-groups in SA_h.

After the bucket sort that forms SA_1, MM computes ISA_1 by specifying as the h-rank of each suffix i in SA_1 the leftmost position in SA_1 (the head) of its group:

	1	2	3	4	5	6	7	8	9	10	11	12
$\boldsymbol{x} =$	a	b	e	a	c	a	d	a	b	e	a	\$
$\mathrm{SA}_1 =$	12	(1	4	6	8	11)	(2	9)	5	7	(3	10)
$\mathrm{ISA}_1 =$	2	7	11	2	9	2	10	2	7	11	2	1

To form SA_2, we consider positive values of $i-1 = \mathrm{SA}_1[j]-h$ for $j = 1, 2, \ldots, n$:

- for $j = 1, 7, 8, 9, 10$, identify in 2-order the suffixes $11, (1, 8), 4, 6$ beginning with a;

- for $j = 11, 12$, identify in 2-order the 2-equal suffixes $(2, 9)$ beginning with b;
- for $j = 3, 6$, identify in 2-order the 2-equal suffixes $(3, 10)$ beginning with e.

Of course groups that are singletons in SA_1 remain singletons in SA_2, and so, after relabeling the groups, we get

$$\begin{array}{rcccccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ \mathrm{SA}_2 = & 12 & 11 & (1 & 8) & 4 & 6 & (2 & 9) & 5 & 7 & (3 & 10) \\ \mathrm{ISA}_2 = & 3 & 7 & 11 & 5 & 9 & 6 & 10 & 3 & 7 & 11 & 2 & 1 \end{array}$$

To form SA_4, we consider positive values of $i-2 = \mathrm{SA}_2[j]-h$ for $j = 1, 2, \ldots, n$:

- for $j = 11, 12$, we identify in 4-order the 4-equal suffixes $(1, 8)$ beginning with ab;
- for $j = 2, 5$, we identify in 4-order the 4-distinct suffixes $9, 2$ beginning with be;
- for $j = 1, 9$, we identify in 4-order the 4-distinct suffixes $10, 3$ beginning with ea.

Hence:

$$\begin{array}{rcccccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ \mathrm{SA}_4 = & 12 & 11 & (1 & 8) & 4 & 6 & 9 & 2 & 5 & 7 & 10 & 3 \\ \mathrm{ISA}_4 = & 3 & 8 & 12 & 5 & 9 & 6 & 10 & 3 & 7 & 11 & 2 & 1 \end{array}$$

The final $\mathrm{SA} = \mathrm{SA}_8$ and $\mathrm{ISA} = \mathrm{ISA}_8$ are achieved after one further doubling that separates the *abea*'s $(1, 8)$ into $8, 1$.

Algorithm MM is complicated by the requirement to keep track of the head of each h-group, but can nevertheless be implemented using as few as $4n$ bytes of storage, in addition to that required for $\boldsymbol{x}$ and SA. It can be represented conceptually as shown in Figure 3.

A time- and space-efficient implementation of MM is available at [M97].

```
h ← 1
initialize SA_1, ISA_1
while some h-group not a singleton
  for j ← 1 to n do
    i ← SA_h[j]−h
    if i > 0 then
      q ← head[h-group[i]]
      SA_2h[q] ← i
      head[h-group[i]] ← q+1
  compute ISA_2h  — update 2h-groups
  h ← 2h
```

Figure 3: Algorithm MM

Larsson & Sadakane [S98, LS99]

After using TSQS to form SA_1, Algorithm LS computes ISA_1 using the *rightmost* (rather than, as in Algorithm MM, the leftmost) position of each group in SA_1 to identify h-rank$[i]$.

	1	2	3	4	5	6	7	8	9	10	11	12
$\boldsymbol{x} =$	a	b	e	a	c	a	d	a	b	e	a	\$
$SA_1 =$	12	(1	4	6	8	11)	(2	9)	5	7	(3	10)
$ISA_1 =$	6	8	12	6	9	6	10	6	8	12	6	1

In addition to identifying h-groups in SA_h that are not singletons, LS also identifies ***runs*** of consecutive positions that are singletons (fully sorted). For this purpose an array $L = L[1..n]$ is maintained, in which $L[j] = \ell$ (respectively, $-\ell$) if and only if j is the leftmost position in SA_h of an h-group (respectively, run) of length ℓ:

	1	2	3	4	5	6	7	8	9	10	11	12
$L =$	-1	5					2		-2		2	

Left-to-right processing of L thus allows runs to be skipped and non-singleton h-groups to be identified, in time proportional to the total number of runs and h-groups. TSQS is again used to sort the suffixes i in each of the identified h-groups according to keys $ISA_h[i+h]$, thus yielding, by Observation 1, a collection of subgroups and subruns in $2h$-order. A straightforward update of L and ISA then yields stage $2h$:

	1	2	3	4	5	6	7	8	9	10	11	12
$SA_2 =$	12	11	(1	8)	4	6	(2	9)	5	7	(3	10)
$ISA_2 =$	4	8	12	5	9	6	10	4	8	12	2	1
$L =$	-2		2		-2		2		-2		2	

A further doubling yields

	1	2	3	4	5	6	7	8	9	10	11	12
$SA_4 =$	12	11	(1	8)	4	6	9	2	5	7	10	3
$ISA_4 =$	4	8	12	5	9	6	10	4	7	11	2	1
$L =$	-2		2		-8							

and then the final results SA_8 and ISA_8 are achieved as for Algorithm MM, with $L[1] = -12$.

Observe that, like MM, LS maintains $ISA_{2h}[i] = ISA_h[i]$ for every suffix i that is a singleton in its h-group. However, unlike MM, LS avoids having to process every position in SA_h (see the **for** loop in Figure 3) by virtue of its use of the array L — in fact, once for some h, i is identified as a singleton, $SA_h[i]$ is never accessed again.

We now remark that in fact L can be eliminated! L is not required to determine non-singleton h-groups because for every suffix i in such a group, $ISA_h[i]$ is by definition the rightmost position in the group. Thus, in particular, at the leftmost position j of the h-group, where $i = SA_h[j]$, we can compute the length ℓ of the group from $\ell = ISA_h[i]-j+1$. Of course L also keeps track of runs of fully sorted suffixes in SA_h,

but, as just remarked, positions in SA_h corresponding to such runs are thereafter unused — it turns out that they can be recycled to perform the run-tracking role. This implementation requires that SA_h be reconstructed from ISA_h in order to provide the final output, a straightforward procedure (see Section 2).

Algorithm LS thus requires $4n$ additional bytes of storage (the integer array ISA), just like MM. As shown in [LS99], LS executes in $O(n \log n)$ time, again the same as MM; however, in practice its running time is usually several times faster.

3.2 Recursive Algorithms [F97]

In this section we consider a family of algorithms that were all discovered in 2003 or later, that are recursive in nature, and that generally execute in worst-case time linear in string length. All are based on an idea first put forward by Farach [F97] for linear-time suffix *tree* construction of strings on an indexed alphabet: they depend on an initial assignment of ***type*** to each suffix (position) in $\boldsymbol{x}$ that separates the suffixes into two or more classes. Thus the recursion in all cases is based on a ***split*** of the given string $\boldsymbol{x} = \boldsymbol{x}^{(0)}$ into disjoint (or almost disjoint) components (subsequences) that are transformed into strings we call $\boldsymbol{x}^{(1)}$ and $\boldsymbol{y}^{(1)}$, chosen so that, if $\mathrm{SA}_{\boldsymbol{x}^{(1)}}$ is (recursively) computed, then in linear time

- $\mathrm{SA}_{\boldsymbol{x}^{(1)}}$ can be used to ***induce*** construction of $\mathrm{SA}_{\boldsymbol{y}^{(1)}}$, and furthermore
- $\mathrm{SA}_{\boldsymbol{x}^{(0)}}$ can then also be computed by a ***merge*** of $\mathrm{SA}_{\boldsymbol{x}^{(1)}}$ and $\mathrm{SA}_{\boldsymbol{y}^{(1)}}$.

Thus the computation of $\mathrm{SA}_{\boldsymbol{x}^{(0)}}$ (in general, $\mathrm{SA}_{\boldsymbol{x}^{(i)}}$) is reduced to the computation of $\mathrm{SA}_{\boldsymbol{x}^{(1)}}$ (in general, $\mathrm{SA}_{\boldsymbol{x}^{(i+1)}}$). To make this strategy efficient and effective, two requirements need to be met.

1. At each recursive step, ensure that

   $$|\boldsymbol{x}^{(i+1)}| / |\boldsymbol{x}^{(i)}| \leq f < 1;$$

   thus the sum of the lengths of the strings processed by all recursive steps is

   $$|\boldsymbol{x}|(1+f+f^2+\ldots) < |\boldsymbol{x}|/\big(1-f\big).$$

   In fact, over all the algorithms proposed so far, $f \leq 2/3$, so that the sum of the lengths is guaranteed to be less than $3|\boldsymbol{x}|$ — for most of them $\leq 2|\boldsymbol{x}|$.

2. Devise an approximate suffix-sorting procedure, ***semisort*** say, that for some sufficiently short string $\boldsymbol{x}^{(i+1)}$ will yield a complete sort of its suffixes and thus terminate the recursion, allowing the suffixes of $\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(i-1)}, \ldots, \boldsymbol{x}^{(0)}$ all to be sorted in turn. Ensure moreover that the time required for *semisort* is linear in the length of the string being processed.

Clearly suffix-sorting algorithms satisfying the above description will compute $\mathrm{SA}_{\boldsymbol{x}}$ (or equivalently $\mathrm{ISA}_{\boldsymbol{x}}$) of a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ in $\Theta(n)$ time. The structure of such algorithms is shown in Figure 4.

All of the algorithms discussed in this subsection compute $\boldsymbol{x'}$ (that is, $\boldsymbol{x}^{(1)}$) and $\boldsymbol{y}$ (that is, $\boldsymbol{y}^{(1)}$) from $\boldsymbol{x}$ (that is, $\boldsymbol{x}^{(0)}$) in similar ways: the alphabet of the split strings

```
procedure construct(x; SA)
split(x; x', y)
semisort(x'; ISA')
if ISA' contains duplicate ranks then
   construct(ISA'; SA_x = SA')
else
   invert(ISA_x = ISA'; SA_x')
induce(SA_x', ISA_x'; SA_y)
merge(SA_x', SA_y; SA_x)
```

Figure 4: General algorithm for recursive SA construction

is in fact the set of suffixes (positions) $1..n$ in $\boldsymbol{x}$, so that $\boldsymbol{x'}$ and $\boldsymbol{y}$ together form a permutation of $1..n$.

Attention then focuses on computing the ranks of the suffixes (positions) i of $\boldsymbol{x}$ that occur in $\boldsymbol{x'}$: we call this sequence (string) of ranks $\mathrm{ISA}_{\boldsymbol{x'}}$, where for $j = 1, 2, \ldots, |\boldsymbol{x'}|$, $\mathrm{ISA}_{\boldsymbol{x'}}[i]$ gives the rank of suffix $i = \boldsymbol{x'}[j]$ of $\boldsymbol{x}$.

Procedure *semisort* computes an approximation ISA' of $\mathrm{ISA}_{\boldsymbol{x'}}$ that ultimately, at some level of recursion, becomes exact — and so we may write $\mathrm{ISA}_{\boldsymbol{x'}} = \mathrm{ISA}'$, then *invert* $\mathrm{ISA}_{\boldsymbol{x'}}$ to form $\mathrm{SA}_{\boldsymbol{x'}}$.

If however ISA' is not exact, then it is used as the input string for a recursive call of the *construct* procedure, thus yielding the suffix array, SA' say, of ISA' — the key observation made here, common to all the recursive algorithms, is that since SA' is the suffix array for the (approximate) ranks of the suffixes identified by $\boldsymbol{x'}$, it is therefore the suffix array for those suffixes themselves. We may accordingly write $\mathrm{SA}_{\boldsymbol{x'}} = \mathrm{SA}'$.

In our discussion below of these algorithms, we focus on the nature of *split* and *semisort* and their consequences for the *induce* and *merge* procedures.

Ko & Aluru [KA03]

Algorithm KA's *split* procedure assigns suffixes $i < n$ in left-to-right order to a sequence $\mathcal{S}$ (respectively, $\mathcal{L}$) iff $\boldsymbol{x}[i..n] <$ (respectively, $>$) $\boldsymbol{x}[i+1..n]$. Suffix n (\$) is assigned to both $\mathcal{S}$ and $\mathcal{L}$. Since $\boldsymbol{x}[i] = \boldsymbol{x}[i{+}1]$ implies that suffixes i and $i{+}1$ belong to the same sequence, it follows that the KA *split* requires time linear in $\boldsymbol{x}$.

Then $\boldsymbol{x'}$ is formed from the sequence of suffixes of smaller cardinality, $\boldsymbol{y}$ from the sequence of larger cardinality. Hence for KA, $|\boldsymbol{x'}| \leq |\boldsymbol{x}|/2$.

For example,

	1	2	3	4	5	6	7	8	9	10	11	12
$\boldsymbol{x} =$	b	a	d	d	a	d	d	a	c	c	a	\$
type $=$	L	S	L	L	S	L	L	S	L	L	L	S/L

yields $|\mathcal{S}| = 4$, $|\mathcal{L}| = 9$, $\boldsymbol{x'} = 2\,5\,8\,12$, $\boldsymbol{y} = 1\,3\,4\,6\,7\,9\,10\,11\,12$.

For every $j \in 1..|\boldsymbol{x'}|$, KA's *semisort* procedure forms $i = \boldsymbol{x'}[j]$, $i_1 = \boldsymbol{x'}[j+1]$ ($i_1 = \boldsymbol{x'}[j]$ if $j = |\boldsymbol{x'}|$), and then performs a radix sort on the resulting substrings $\boldsymbol{x}[i..i_1]$, a calculation that requires $\Theta(n)$ time. The result of this sort is a ranking ISA'

of the substrings $\boldsymbol{x}[i..i_1]$, hence an approximate ranking of the suffixes (positions) $i = \boldsymbol{x'}[j]$. In our example, *semisort* yields

	1	2	3	4	5	6	7	8	9	10	11	12
$\boldsymbol{x} =$	b	a	d	d	a	d	d	a	c	c	a	\$
$\boldsymbol{x'} =$		2			5			8				12
$\mathrm{ISA}' =$		3			3			2				1

If after *semisort* the entries (ranks) in ISA' are distinct, then a complete ordering of the suffixes of $\boldsymbol{x'}$ has been computed ($\mathrm{ISA}' = \mathrm{ISA}_{\boldsymbol{x'}}$); if not, then as indicated in Figure 4, the *construct* procedure is recursively called on ISA'. In our example, one recursive call suffices for a complete ordering $(12, 8, 5, 2)$ of the suffixes of $\boldsymbol{x'}$, yielding $\mathrm{ISA}_{\boldsymbol{x'}} = 4321$.

At this point KA deviates from the pattern of Figure 4 in two ways: it combines the *induce* and *merge* procedures into a single KA-merge (see Figure 5), and it computes $\mathrm{SA}_{\boldsymbol{x}}$ directly without reference to $\mathrm{ISA}_{\boldsymbol{x}}$ [1].

```
initialize SA ← SA_1, head[1..α], tail[1..α]
for i ← |x'| downto 1 do
   λ ← x[x'[i]]
   SA[tail[λ]] ← x'[i]
   tail[λ] ← tail[λ]−1
for j ← 1 to n do
   i ← SA[j]
   if type[i−1] = L then
      λ ← x[i−1]
      SA[head[λ]] ← i−1
      head[λ] ← head[λ]+1
```

Figure 5: Algorithm KA-merge

First SA_1 is computed, yielding 1-groups for which the leftmost and rightmost positions are specified in arrays $\mathrm{head}[1..\alpha]$ and $\mathrm{tail}[1..\alpha]$, respectively. Since in each 1-group all the S-suffixes are lexicographically greater than all the L-suffixes, and since the S-suffixes have been sorted, KA-*merge* can place all the S-suffixes in their final positions in SA — each time this is done, the tail for the current group is decremented by one. (In this description, we assume that $|\mathcal{S}| \leq |\mathcal{L}|$; obvious adjustments yields a corresponding approach for the case $|\mathcal{L}| < |\mathcal{S}|$.)

The SA at this stage is shown below, with "$-$" denoting an empty position:

	1	2	3	4	5	6	7	8	9	10	11	12
$\mathrm{SA} =$	12	$(-$	8	5	2)	$(-)$	$(-$	$-)$	$(-$	$-$	$-$	$-)$
$\mathrm{type} =$	S	L	S	S	S	L	L	L	L	L	L	L

To sort the L-suffixes, we scan SA left to right. For each suffix position $i = \mathrm{SA}[j]$ that we encounter in the scan, if $i-1$ is an L-suffix still awaiting sorting (not yet placed in the SA), we place $i-1$ at the head of its group in SA and increment the

[1] In [KA03] it is claimed that the ISA must be built in unison with the SA for this procedure to work, but we have found that this is actually unnecessary.

head of the group by one. Suffix $i-1$ is now sorted and will not be moved again. The correctness of this procedure depends on the fact that when the scan of SA reaches position j, SA[j] is already in its final position. In our example, placements begin when $j = 1$, so that $i = \text{SA}[1] = 12$. Since suffix $i-1 = 11$ is type L, it is placed at the front of the a group (of which it happens to be the only member):

$$\begin{array}{rcccccccccccc} & {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 4} & {\scriptstyle 5} & {\scriptstyle 6} & {\scriptstyle 7} & {\scriptstyle 8} & {\scriptstyle 9} & {\scriptstyle 10} & {\scriptstyle 11} & {\scriptstyle 12} \\ \text{SA} = & 12 & (11 & 8 & 5 & 2) & (-) & (- & -) & (- & - & - & -) \\ \text{type} = & S & L & S & S & S & L & L & L & L & L & L & L \end{array}$$

Next the scan reaches $j = 2$, $i = \text{SA}[2] = 11$, and we place $i-1 = 10$ at the front of the c group at SA[7] and increment the group head.

$$\begin{array}{rcccccccccccc} & {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 4} & {\scriptstyle 5} & {\scriptstyle 6} & {\scriptstyle 7} & {\scriptstyle 8} & {\scriptstyle 9} & {\scriptstyle 10} & {\scriptstyle 11} & {\scriptstyle 12} \\ \text{SA} = & 12 & (11 & 8 & 5 & 2) & (-) & (10 & -) & (- & - & - & -) \\ \text{type} = & S & L & S & S & S & L & L & L & L & L & L & L \end{array}$$

The scan continues until finally

$$\begin{array}{rcccccccccccc} & {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 4} & {\scriptstyle 5} & {\scriptstyle 6} & {\scriptstyle 7} & {\scriptstyle 8} & {\scriptstyle 9} & {\scriptstyle 10} & {\scriptstyle 11} & {\scriptstyle 12} \\ \text{SA} = & 12 & 11 & 8 & 5 & 2 & 1 & 10 & 9 & 7 & 4 & 6 & 3 \end{array}$$

Algorithm KA can be implemented to use only $4n$ bytes plus $1.25n$ bits in addition to the storage required for $\boldsymbol{x}$ and SA.

Kärkkäinen & Sanders [KS03]

The *split* procedure of Algorithm KS first separates the suffixes i of $\boldsymbol{x}$ into sequences $\mathcal{S}_1$ (every third suffix in $\boldsymbol{x}$: $i \equiv 1 \bmod 3$) and $\mathcal{S}_{02}$ (the remaining suffixes: $i \not\equiv 1 \bmod 3$). Thus in this algorithm three types $0, 1, 2$ are identified: $\boldsymbol{x'}$ is formed from $\mathcal{S}_{02}$ by

$$\boldsymbol{x'} = (i \equiv 2 \bmod 3)\ (i \equiv 0 \bmod 3),$$

while $\boldsymbol{y}$ is formed directly from $\mathcal{S}_1$. For our example string

$$\begin{array}{rcccccccccccc} & {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 4} & {\scriptstyle 5} & {\scriptstyle 6} & {\scriptstyle 7} & {\scriptstyle 8} & {\scriptstyle 9} & {\scriptstyle 10} & {\scriptstyle 11} & {\scriptstyle 12} \\ \boldsymbol{x} = & b & a & d & d & a & d & d & a & c & c & a & \$ \end{array}$$

we find $\boldsymbol{x'} = (2\,5\,8\,11)(3\,6\,9\,12)$, $\boldsymbol{y} = 1\,4\,7\,10$. Note that $|\boldsymbol{x'}| \le \lfloor 2|\boldsymbol{x}|/3 \rfloor$.

Construction of ISA$'$ using *semisort* begins with a linear-time 3-sort of suffixes $i \in \mathcal{S}_{02}$ based on triples $t_i = \boldsymbol{x}[i..i+2]$. Thus a 3-order of these suffixes is established for which a 3-rank r_i can be computed, as illustrated by our example:

i	2	3	5	6	8	9	11	12
t_i	*add*	*dda*	*add*	*dda*	*acc*	*cca*	*a*\$−	\$−−
r_i	4	6	4	6	3	5	2	1

These ranks enable ISA$'$ to be formed for $\boldsymbol{x'}$:

$$\begin{array}{rcccccccc} & {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 4} & {\scriptstyle 5} & {\scriptstyle 6} & {\scriptstyle 7} & {\scriptstyle 8} \\ \text{ISA}' = & (4 & 4 & 3 & 2) & (6 & 6 & 5 & 1) \end{array}$$

As with Algorithm KA, one recursive call on $\boldsymbol{x'} = 44326651$ suffices to complete the ordering, yielding $\mathrm{ISA}_{\boldsymbol{x'}} = 54328761$ — this gives the ordinal ranks in $\boldsymbol{x}$ of the suffixes $\boldsymbol{x'} = 2\,5\,8\,11\,3\,6\,9\,12$.

The *induce* procedure sorts the suffixes specified by $\boldsymbol{y}$ based on the ordering $\mathrm{ISA}_{\boldsymbol{x'}}$. First $\mathrm{SA}_{\boldsymbol{x'}} = 12\,11\,8\,5\,2\,9\,6\,3$ is formed by linear-time processing of $\mathrm{ISA}_{\boldsymbol{x'}}$. Then a left-to-right scan of $\mathrm{SA}_{\boldsymbol{x'}}$ allows us to identify suffixes $i \equiv 2 \bmod 3$ in increasing order of rank and thus to select letters $\boldsymbol{x}[i{-}1]$, $i{-}1 \equiv 1 \bmod 3$, in the same order. A stable bucket sort of these letters will then provide the suffixes of $\boldsymbol{y}$ in increasing lexorder. In our example $\mathrm{SA}_{\boldsymbol{x'}}[2..5] = 11\,8\,5\,2$, and so we consider $\boldsymbol{x}[10] = c$, $\boldsymbol{x}[7] = \boldsymbol{x}[4] = d$, $\boldsymbol{x}[1] = b$. A stable sort yields *bcdd* corresponding to $\mathrm{SA}_{\boldsymbol{y}} = 1\,10\,7\,4$.

Thus we may suppose that $\mathrm{SA}_{\boldsymbol{x'}}$ and $\mathrm{SA}_{\boldsymbol{y}}$ are both in sorted order of suffix. The KS *merge* procedure may then be thought of as a straightforward merge of these two strings into the output array $\mathrm{SA}_{\boldsymbol{x}}$, where at each step we need to decide in constant time whether suffix i_{02} of $\mathrm{SA}_{\boldsymbol{x'}}$ is greater or less than suffix i_1 of $\mathrm{SA}_{\boldsymbol{y}}$. Observing that $i_1{+}1 \equiv 2 \bmod 3$ and $i_1{+}2 \equiv 0 \bmod 3$, we identify two cases:

- if $i_{02} \equiv 2 \bmod 3$, $i_{02}+1 \equiv 0 \bmod 3$, and so it suffices to compare the pairs $\big(\boldsymbol{x}[i_{02}], \mathrm{rank}(i_{02}{+}1)\big)$ and $\big(\boldsymbol{x}[i_1], \mathrm{rank}(i_1{+}1)\big)$;
- if $i_{02} \equiv 0 \bmod 3$, $i_{02}+2 \equiv 2 \bmod 3$, and so it suffices to compare the triples $\big(\boldsymbol{x}[i_{02}..i_{02}{+}1], \mathrm{rank}(i_{02}{+}2)\big)$ and $\big(\boldsymbol{x}[i_1..i_1{+}1], \mathrm{rank}(i_1{+}2)\big)$.

We now observe that each of the ranks required by these comparisons is available in constant time from $\mathrm{ISA}_{\boldsymbol{x'}}$! For if $i \equiv 2 \bmod 3$, then

$$\mathrm{rank}(i) = \mathrm{ISA}_{\boldsymbol{x'}}\big[\lfloor (i{+}1)/3 \rfloor\big],$$

while if $i \equiv 0 \bmod 3$, then

$$\mathrm{rank}(i) = \mathrm{ISA}_{\boldsymbol{x'}}\big[\lfloor (n{+}1)/3 \rfloor + \lfloor i/3 \rfloor\big].$$

Thus the merge of the two lists requires $\Theta(n)$ time.

Excluding $\boldsymbol{x}$ and SA, Algorithm KS can be implemented in $6n$ bytes of working storage. A recent variant of KS [N05] permits construction of a succinct suffix array in $O(n)$ time using only $O(n \log\sigma \log^q_\sigma n)$ bits of working memory, where $q = \log_2 3$.

Kim, Jo & Park [KSPP03, HSS03, KJP04]

The KJP *split* procedure adopts the same approach as Farach's suffix tree construction algorithm [F97]: it forms $\boldsymbol{x'}$, the string of odd suffixes (positions) $i \equiv 1 \bmod 2$ in $\boldsymbol{x}$, and the corresponding string $\boldsymbol{y}$ of even positions. $\mathrm{ISA}_{\boldsymbol{x'}}$ is then formed by a recursive sort of the suffixes identified by $\boldsymbol{x'}$. Algorithm KJP is not quite linear in its operation, running in $O(n\mathrm{loglog}n)$ worst-case time.

For KJP we modify our example slightly to make it more illustrative:

$$\begin{array}{rc} & \begin{array}{cccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \end{array} \\ \boldsymbol{x} = & \begin{array}{cccccccccccc} b & a & d & d & d & d & a & c & c & a & \$ \end{array} \end{array}$$

yielding $\boldsymbol{x'} = 1\,3\,5\,7\,9\,11$, $\boldsymbol{y} = 2\,4\,6\,8\,10$.

The KJP *semisort* 2-sorts prefixes $p_i = \boldsymbol{x}[i..i{+}1]$ of each odd suffix i and assigns to each an ordinal rank r_i:

i	11	7	1	9	3	5
p_i	\$−	ac	ba	ca	dd	dd
r_i	1	2	3	4	5	5

As in the other recursive algorithms, a new string ISA' is formed from these ranks; in our example,

$$\begin{array}{rc} & \scriptstyle 1\ 2\ 3\ 4\ 5\ 6 \\ \mathrm{ISA}' = & 3\ 5\ 5\ 2\ 4\ 1 \end{array}$$

As with the other recursive algorithms, one recursive call suffices to find $\mathrm{ISA}_{\boldsymbol{x'}} = 365241$ corresponding to $\boldsymbol{x'} = 1\,3\,5\,7\,9\,11$. At this point KJP computes the inverse array $\mathrm{SA}_{\boldsymbol{x'}} = 11\,7\,1\,9\,5\,3$. The KJP *induce* procedure can now compute $\mathrm{SA}_{\boldsymbol{y}}$, the sorted list of even suffixes, in a straightforward manner: first set $\mathrm{SA}_{\boldsymbol{y}}[i] \leftarrow \mathrm{SA}_{\boldsymbol{x'}}[i]-1$, and then sort $\mathrm{SA}_{\boldsymbol{y}}$ stably, using $\boldsymbol{x}\big[\mathrm{SA}_{\boldsymbol{y}}[i]\big]$ as the sort key for suffix $\mathrm{SA}_{\boldsymbol{y}}[i]$:

$$\begin{array}{rc} & \scriptstyle 1\ \ 2\ 3\ 4\ 5\ 6 \\ \mathrm{SA}_{\boldsymbol{x'}} = & 11\ 7\ 1\ 9\ 5\ 3 \\ \mathrm{SA}_{\boldsymbol{y}} = & 10\ 2\ 8\ 6\ 4 \end{array}$$

The KJP *merge* is more complex. In order to merge $\mathrm{SA}_{\boldsymbol{x'}}$ and $\mathrm{SA}_{\boldsymbol{y}}$ efficiently, we need to compute an array $C\big[1..\lceil n/2\rceil\big]$, in which $C[i]$ gives the number of suffixes in $\mathrm{SA}_{\boldsymbol{x'}}$ that lie between $\mathrm{SA}_{\boldsymbol{y}}[i]$ and $\mathrm{SA}_{\boldsymbol{y}}[i-1]$ in the final SA (with special attention to end conditions $i = 1$ and $i = |\boldsymbol{y}|+1$). In [KJP04] it is explained how C can be computed in $\log|\boldsymbol{x'}|$ time using a suffix array search (pattern-matching) algorithm described in [SKPP03]. We omit the details, however, for our example we would find

$$\begin{array}{rc} & \scriptstyle 1\ 2\ 3\ 4\ 5\ 6 \\ \mathrm{C} = & 0\ 1\ 1\ 0\ 1\ 1 \end{array}$$

With C in hand, merging is just a matter of using each $C[i]$ to count how many consecutive $\mathrm{SA}_{\boldsymbol{x'}}$ entries to insert between consecutive $\mathrm{SA}_{\boldsymbol{y}}$ entries.

There are two other algorithms which, like KJP, perform an odd/even split of the suffixes. Algorithm KSPP [KSPP03] was the first of these, and although its worst-case execution time is $\Theta(n)$, it is generally considered to be of only theoretical interest, mainly due to high memory requirements. On the other hand, Algorithm HSS [HSS03] uses "succinct data structures" [M99] effectively to construct a (succinct) suffix array in $O(n \log\log\sigma)$ time with only $\Theta(n\log\sigma)$ bits of working memory. (Compare the variant [N05] of Algorithm KS mentioned above.) It is not clear how practical these lightweight approaches are, since their succinctness may well adversely affect speed.

3.3 Induced Copying Algorithms [BW94]

The algorithms in this class are arguably the most diverse of the three main divisions of SACAs discussed in this paper. They are united by the idea that a (usually) complete sort of a selected subset of suffixes can be used to ***induce*** a fast sort of the remaining suffixes. This induced sort is similar to the *induce* procedures employed in the recursive SACAs; the difference is that some sort of iteration is used in place of the recursion. This replacement (of recursion by iteration) probably largely explains why several of the induced copying algorithms are faster in practice than

any of the recursive algorithms (as we shall discover in Section 4), eventhough none of these algorithms is linear in the worst case. In fact, their worst-case asymptotic complexity is generally $O(n^2 \log n)$. In terms of space requirements, these algorithms are lightweight: for many of them, use of additional working storage amounts to less than n bytes.

We begin with brief outlines of the induced copying algorithms:

- Itoh & Tanaka [IT99] select suffixes i of "type B" — those satisfying $\boldsymbol{x}[i] \leq \boldsymbol{x}[i+1]$ — for complete sorting, thus inducing a sort of the remaining suffixes.
- Seward [S00] on the other hand sorts certain 1-groups, using the results to induce sorts of corresponding 2-groups, an approach that also forms the basis of Algorithms MF [MF04] and SS [SS05].
- A third approach, due to Burkhardt & Kärkkäinen, uses a small integer h to form a "sample" S of suffixes that is then h-sorted; using a technique reminiscent of the recursive algorithms, the resulting h-ranks are then used to induce a complete sort of all the suffixes.
- Finally, the as-yet-unpublished algorithm of Maniscalco [M05] computes $\mathrm{ISA}_{\boldsymbol{x}}$ using an iterative technique that, beginning with 1-groups, uses h-groups to induce the formation of $(h+1)$-groups.

Itoh & Tanaka [IT99]

Algorithm IT classifies each suffix i of $\boldsymbol{x}$ as being type A if $x[i] > x[i+1]$ or type B if $x[i] \leq x[i+1]$ (compare types L and S of Algorithm KA). The key observation of Itoh and Tanaka is that once all the groups of type B suffixes are sorted, the order of the type A suffixes is easy to derive. For example:

	1	2	3	4	5	6	7	8	9	10	11	12
$\boldsymbol{x} =$	b	a	d	d	a	d	d	a	c	c	a	\$
type $=$	A	B	B	A	B	B	A	B	B	A	A	B

To form the full SA, we begin by computing the 1-group boundaries, noting the beginning and end of each 1-group with arrays head$[1..\sigma]$ and tail$[1..\sigma]$ (recall $\sigma = |\Sigma|$). Each 1-group is further partitioned into two portions, so that in the first portion there is room for the type A suffixes, and in the second for the type B suffixes. For each group the position of the A/B partition is recorded. Observe that within a 1-group, type A suffixes should always come before type B suffixes. The SA at this stage is shown below, with "–" denoting an empty position:

	1	2	3	4	5	6	7	8	9	10	11	12
SA $=$	12	(–	2	5	8)	(–)	(–	9)	(–	–	3	6)
type $=$	B	A	B	B	B	A	A	B	A	A	A	A

Algorithm IT now sorts the B suffixes using a fast string sorting algorithm. In [IT99] multikey quicksort (MKQS) [BS97] is proposed, but any other fast sort, such as burst sort [SZ04] or the elaborate approach introduced in Algorithm MF (see below), could be used:

$$\begin{array}{rcccccccccccc} & {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 4} & {\scriptstyle 5} & {\scriptstyle 6} & {\scriptstyle 7} & {\scriptstyle 8} & {\scriptstyle 9} & {\scriptstyle 10} & {\scriptstyle 11} & {\scriptstyle 12} \\ \text{SA} = & 12 & (- & 8 & 5 & 2) & (-) & (- & 9) & (- & - & 6 & 3) \\ \text{type} = & B & A & B & B & B & A & A & B & A & A & A & A \end{array}$$

To sort the A-suffixes, and complete the SA, we scan SA left to right, $j = 1, 2, \ldots, n$. For each suffix position $i = \text{SA}[j]$ that we encounter in the scan, if $i-1$ is an A-suffix still awaiting sorting (that is, it has not yet been placed in the SA), then we place $i-1$ at the head of its group in SA and increment the head of the group by one. Suffix $i-1$ is now sorted and will not be moved again. Like Algorithm KA, the correctness of this procedure depends on $\text{SA}[j]$ already being in its final position when the scan of SA reaches position j. In our example, placements begin when $j = 1$, $i = \text{SA}[1] = 12$. Suffix $i-1 = 11$ is type A, so we place 11 at the front of the a group (of which it happens to be the only unsorted member), and it is now sorted:

$$\begin{array}{rcccccccccccc} & {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 4} & {\scriptstyle 5} & {\scriptstyle 6} & {\scriptstyle 7} & {\scriptstyle 8} & {\scriptstyle 9} & {\scriptstyle 10} & {\scriptstyle 11} & {\scriptstyle 12} \\ \text{SA} = & 12 & 11 & 8 & 5 & 2 & (-) & (- & 9) & (- & - & 6 & 3) \\ \text{type} = & B & A & B & B & B & A & A & B & A & A & A & A \end{array}$$

Next the scan reaches $j = 2$, $i = \text{SA}[2] = 11$, and so we place $i-1 = 10$ at the front of its c group at SA[7] and increment the group head, completing that group:

$$\begin{array}{rcccccccccccc} & {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 4} & {\scriptstyle 5} & {\scriptstyle 6} & {\scriptstyle 7} & {\scriptstyle 8} & {\scriptstyle 9} & {\scriptstyle 10} & {\scriptstyle 11} & {\scriptstyle 12} \\ \text{SA} = & 12 & 11 & 8 & 5 & 2 & (-) & 10 & 9 & (- & - & 6 & 3) \\ \text{type} = & B & A & B & B & B & A & A & B & A & A & A & A \end{array}$$

The scan continues, eventually arriving at the final SA :

$$\begin{array}{rcccccccccccc} & {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 4} & {\scriptstyle 5} & {\scriptstyle 6} & {\scriptstyle 7} & {\scriptstyle 8} & {\scriptstyle 9} & {\scriptstyle 10} & {\scriptstyle 11} & {\scriptstyle 12} \\ \text{SA} = & 12 & 11 & 8 & 5 & 2 & 1 & 10 & 9 & 7 & 4 & 6 & 3 \end{array}$$

Figure 6 gives an algorithm capturing these ideas. The attentive reader will note the similarity between it and Algorithm KA (Subsection 3.2). In fact, the set of B-suffixes used in Algorithm IT is a superset of the S-suffixes treated in Algorithm KA.

```
initialize SA ← SA_1
  — head[1..σ] and tail[1..σ] mark 1-group boundaries
  — part[1..σ] marks A/B partition of each 1-group
for h ← 1 to σ do
  suffixsort(SA[part[h]], SA[part[h]+1], ..., SA[tail[h]])
for j ← 1 to n do
  i ← SA[j]
  if type[i−1] = A then
    λ ← x[i−1]
    SA[head[λ]] ← i−1
    head[λ] ← head[λ]+1
```

Figure 6: Algorithm IT

Clearly IT executes in time linear in n except for the up to σ suffix sorts of the possibly $\Theta(n)$ B-suffixes in each 1-group; these sorts may require $O(n^2 \log n)$ time in pathological cases. In practice, however, IT is quite fast. It is also lightweight: with careful implementation (for example, both head and tail arrays do not need to be stored, and *suffixsort* can be executed in place), IT requires less than n bytes of additional working storage when n is large (megabytes or more) with respect to σ.

Seward [S00]

Algorithm S begins with a linear-time 2-sort of the suffixes of $\boldsymbol{x}$, thus forming SA_2 in which the boundaries of each 2-group are identified by the head array — also used to mark boundaries between the 1-groups. Therefore in this case head = head$[1..\sigma, 1..\sigma]$, allowing access to every boundary head$[\lambda, \mu]$ for every $\lambda, \mu \in \Sigma$. For our example the result of the 2-sort could be represented as follows:

	1	2	3	4	5	6	7	8	9	10	11	12
$\boldsymbol{x} =$	b	a	d	d	a	d	d	a	c	c	a	\$
$\mathrm{SA}_2 =$	12	(11	8	[2	5])	1	(10	9)	([4	7]	[3	6])

where () encloses non-singleton 1-groups, [ ] encloses non-singleton 2-groups.

Now consider a 1-group G_λ corresponding to a common single-letter prefix λ. Suppose that the suffixes of G_λ are fully sorted, yielding a sequence G^*_λ in ascending lexorder. Imagine now that G^*_λ is traversed in lexorder: for every suffix $i > 1$, the suffix $i-1$ can be placed in its final position in $\mathrm{SA}_{\boldsymbol{x}}$ at the head of the 2-group for $\boldsymbol{x}[i-1]\lambda$ — provided head$\big[\boldsymbol{x}[i-1], \lambda\big]$ is incremented by one after the suffix is placed there, thus allowing for correct placement of any other suffixes in the same 2-group. The lexorder of G^*_λ ensures that the suffixes $i-1$ also occur in lexorder within each 2-group.

This is essentially the strategy of Algorithm S: it uses an efficient string sort [BM93] to sort completely the unsorted suffixes in a 1-group that currently contains a minimum number of unsorted suffixes, then uses the sorted suffixes i to ***induce*** a sort of suffixes $i-1$. Thus all suffixes can be completely sorted at the cost of a complete sort of only half of them.

The process can be made still more efficient by observing that when G_λ is sorted, the suffixes with prefix λ^2 can be omitted, provided the 2-group corresponding to λ^2 is traversed *after* the traversal of G^*_λ. To see this, suppose there exists a suffix $\lambda^k \mu \boldsymbol{v}$ in G_λ, $k \geq 2, \mu \neq \lambda$. Then the suffix $\lambda\mu\boldsymbol{v}$ will have been sorted into G^*_λ and already processed to place suffix $\boldsymbol{x}[i..n] = \lambda^2\mu\boldsymbol{v}$ at head$[\lambda, \lambda]$. Thus when $\lambda^2\mu\boldsymbol{v}$ is itself processed, suffix $\boldsymbol{x}[i-1]\lambda^2\mu\boldsymbol{v}$ will be placed at head$\big[\boldsymbol{x}[i-1], \lambda\big]$ — this will again be (the now incremented) head$[\lambda, \lambda]$ if $k \geq 3$ ($\boldsymbol{x}[i-1] = \lambda$).

We can apply Algorithm S to our example string:

Iteration 1 The 1-group corresponding to $\lambda =$ \$ contains only the singleton unsorted suffix $i = 12$. Thus the sort is trivial: 12 is already in its final position in SA, and suffix $i-1 = 11$ is put in final position at head$[a, \$] = 2$.

Iteration 2 The minimum 1-group corresponding to b contains only suffix $i = 1$, which is therefore in final position. Since $i-1 = 0$, there is no further action.

Iteration 3 The minimum 1-group corresponds to $\lambda = c$; it again has only one entry to be sorted, since one of the 2-groups represented is cc. Thus suffix $i = 10$ is in final position at head$[c, a] = 7$, and determines the final position of suffix $i-1 = 9$ at head$[c, c] = 8$. Then finally for $i = 9$, the final position of suffix $i-1 = 8$ is fixed at head$[a, c] = 3$.

Iteration 4 The 1-group for $\lambda = a$ now contains only the two unsorted suffixes 2 and 5, since 11 and 8 have been put in final position by previous iterations. The sort yields SA[4] = 5, SA[5] = 2, so that the completely sorted 1-group becomes SA[2..5] = 11 8 5 2. For $i = 11$, suffix $i-1 = 10$ is already in final position; for $i = 8$, suffix $i-1 = 7$ is placed in final position at head$[d, a] = 9$; then, for $i = 5$, after head$[d, a]$ is incremented, suffix $i-1 = 4$ is placed in final position at head$[d, a] = 10$; for $i = 2$, $i-1 = 1$ is already in final position.

Iteration 5 The final group corresponds to $\lambda = d$; by now its only unsorted suffixes, 3 and 6, belong to the 2-group dd and so do not require sorting. As a result of Iteration 4, SA[9..10] = 7 4. Thus, for $i = 7$, suffix $i-1 = 6$ is placed at head$[d, d] = 11$, while for $i = 4$, the final suffix $i-1 = 3$ is placed at head$[d, d] = 12$.

For this example, only one simple sort (of suffixes 2 and 5 in Iteration 4) needs to be performed in order to compute $\mathrm{SA}_{\boldsymbol{x}}$!

Algorithm S shares the $O(n^2 \log n)$ worst case time of other induced copying algorithms, but is nevertheless very fast in practice. However, its running time sometimes seems to degrade significantly when the average lcp, $\overline{\mathrm{lcp}}$, is large, for reasons that are not quite clear. This problem is addressed by a variant, Algorithm MF, discussed next. Like IT, Algorithm S can run using less than n bytes of working storage.

Manzini & Ferragina [MF04]

Algorithm MF is a variant of Algorithm S that replaces TSQS [BM93], used to sort the 2-groups within a selected 1-group, by a more elaborate and sophisticated approach to suffix-sorting. This approach is two-tiered, depending initially on a user-specified integer lcp^*, the longest lcp of a group of suffixes that will be sorted using a standard method. (Typically, for large files, lcp^* will be chosen in the range 500..5000.) Thus, if a 2-group of suffixes is to be sorted, then MKQS [BS97] (rather than TSQS) will be employed until the recursion of MKQS reaches depth lcp^*: if the sort is not complete, this defines a set $I_m = \{i_1, i_2, ..., i_m\}$, $m \geq 2$, of suffixes such that

$$\mathrm{lcp}(i_1, i_2, \ldots, i_m) \geq \mathrm{lcp}^*.$$

At this point, the methodology used to complete the sort of these m suffixes is chosen depending on whether m is "large" or "small".

If m is small, then a sorting method called ***blind sort*** [FG99] is invoked that uses at most 36m bytes of working storage. Therefore, if blind sort is used only for $m \leq n/Q$, its space overhead will be at most $(36/Q)n$ bytes; by choosing $Q \geq 1000$, say — and thus giving special treatment to cases where "not too many" suffixes share

a "long" lcp — it can be ensured that for small m, the space used is a very small fraction of the $5n$ bytes required for $\boldsymbol{x}$ and $\mathrm{SA}_{\boldsymbol{x}}$.

Blind sort of I_m depends on the construction of a ***blind trie*** data structure [FG99]: essentially the strings

$$\boldsymbol{x}[i_j+\mathrm{lcp}^*..n],\ j=1,2,\ldots,m$$

are inserted one-by-one into an initially empty blind trie; then, as explained in [FG99], a left-to-right traversal of the trie obtains the suffixes in lexorder, as required.

If m is large ($> n/Q$), Algorithm MF reverts to the use of a slightly modified TSQS, as in Algorithm S; however, whenever at some recursive level of execution of TSQS a new set of suffixes I'_m is identified for which $m \le n/Q$, then blind sort is again invoked to complete the sort of I'_m.

Following the initial MKQS sort to depth lcp^*, the dual strategy (blind sort/TSQS) described so far to complete the sort is actually only one of two strategies employed by Algorithm MF. Before resorting to the dual strategy, MF tries to make use of ***generalized induced copying***, as we now explain.

Suppose that for $i_1 \in I_m$ and for some least $\ell \in 1..\mathrm{lcp}^*-1$,

$$\boldsymbol{x}[i_1+\ell..i_1+\ell+1] = \lambda\mu,$$

where $[\lambda,\mu]$ identifies a 2-group that as a result of previous processing has already been fully sorted. Since the m suffixes in I_m share a common prefix, it follows that *every* suffix in I_m occurs in the same 2-group $[\lambda,\mu]$. Since moreover the m suffixes in I_m are identical up to position ℓ, it follows that the order of the suffixes in I_m is determined by their order in $[\lambda,\mu]$. Thus if such a 2-group exists, it can be used to "induce" the correct ordering of the suffixes in I_m, as follows:

(1) Bucket-sort the entries $i_j \in I_m$ in ascending order of *position* (not suffix), so membership in I_m can be determined using binary search (step (3)).

(2) Scan the 2-group $[\lambda,\mu]$ to identify a match for suffix $i_1+\ell$, say at some position q.

(3) Scan the suffixes (positions) listed to the left and to the right of q in 2-group $[\lambda,\mu]$; for each suffix i, use binary search to determine whether or not $i-\ell$ occurs in (the now-sorted) I_m. If it does occur, then mark the suffix i in $[\lambda,\mu]$.

(4) When m suffixes have been marked, scan the 2-group $[\lambda,\mu]$ from left to right: for each marked suffix i, copy $i-\ell$ left-to-right into I_m.

Step (2) of this procedure can be time-consuming, since it may involve a $\Theta(n)$-time match of two suffixes; in [MF04] an efficient implementation of step (2) is described that uses only a very small amount of extra space.

Of course if no such ℓ, hence no such 2-group, exists, then this method cannot be used: the dual strategy described above must be used instead.

In practice Algorithm MF runs faster than any of Algorithms KS, IT or S; in common with other induced copying algorithms, it uses less than n bytes of additional working storage but can require as much as $O(n^2 \log n)$ time in the worst case.

Schürmann & Stoye [SS05]

Algorithm SS could arguably be classified as a prefix-doubling algorithm. Certainly it is a hybrid: it first applies a prefix-doubling technique to sort individual h-groups, then uses Seward's induced copying approach to extend the sort to other groups of suffixes.

For SS, the integer h is actually a user-specified parameter, chosen to satisfy $h < \log_\sigma n$. First a radix sort is performed to compute SA_h, then the corresponding ISA_h, in which the h-rank of each h-group is formed from the tail of the h-group in SA_h (the same system used in Algorithm LS). Thus, for example, using $h = 2$, the result of the first phase of processing would be just the same as after the second iteration of LS:

	1	2	3	4	5	6	7	8	9	10	11	12
$\boldsymbol{x} =$	a	b	e	a	c	a	d	a	b	e	a	\$
$\mathrm{SA}_2 =$	12	11	(1	8)	4	6	(2	9)	5	7	(3	10)
$\mathrm{ISA}_2 =$	4	8	12	5	9	6	10	4	8	12	2	1

In its second phase, SS considers h-groups in SA_h that are not singletons. Let H be one such h-group. The observation is made that since every suffix i in H has the same prefix of length h, therefore the order of each i in H is determined by the rank of suffix $i+h$; that is, by $\mathrm{ISA}_h[i+h]$. A sort of all the non-singleton h-groups in SA_h thus leads to the construction of SA_{2h} and ISA_{2h}:

	1	2	3	4	5	6	7	8	9	10	11	12
$\mathrm{SA}_4 =$	12	11	(1	8)	4	6	9	2	5	7	10	3
$\mathrm{ISA}_4 =$	3	8	12	5	9	6	10	3	7	11	2	1

Observe that as a result of the prefix-doubling, the h-groups (2 9) and (3 10) have become completely sorted.

To entries in h-groups that become completely sorted by prefix-doubling, SS applies Algorithm S: if suffix i is in fixed position in SA, then the final position of suffix $i-1$ can also be determined. Thus, in our example, the sort of the h-group (2 9) that yields $2h$-order $9, 2$ induces a corresponding sorted order $8, 1$ for the $2h$-group (1 8), completing the sort.

Algorithm SS iterates this second phase – prefix-doubling followed by induced copying – until all entries in SA are singletons. Note that after the first iteration, the induced copying will as a rule refine the h-groups so that they break down into $(h+k)$-groups for various values of $k \geq 0$; thus, after the first iteration, the prefix-doubling is approximate.

Algorithm SS has worst-case time complexity $O(n^2)$ and appears to be very fast in practice, competitive with Algorithm MF. However, it is not quite lightweight, requiring somewhat more than $4n$ bytes of additional working storage.

Burkhardt & Kärkkäinen [BK03]

In a similar way to the recursive algorithms of Section 3.2, Algorithm BK computes $\mathrm{SA}_{\boldsymbol{x}}$ by first ordering a sample of the suffixes $\mathcal{S}$. The relative ranks of the suffixes in

$\mathcal{S}$ are then used to accelerate a basic string sorting algorithm, such as MKQS [BS97], applied to all the suffixes.

Central to BK is a mathematical construct called a difference cover, which defines the suffixes in $\mathcal{S}$. A difference cover D_h is a set of integers in the range $0..h-1$ such that for all $i \in 0..h-1$, there exist $j, k \in D_h$ such that $i \equiv k - j (\mathrm{mod}\ h)$. For a chosen D_h, $\mathcal{S}$ contains the suffixes of $\boldsymbol{x}$ beginning at positions i such that $i \bmod h \in D_h$.

For example $D_7 = \{1, 2, 4\}$ is a difference cover modulo 7. If we were to sample according to D_7 then for the string

$$\begin{array}{rccccccccccccccccccc} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 \\ \boldsymbol{x} = & b & a & d & d & a & d & d & b & a & d & d & a & d & d & b & a & d & d & \$ \end{array}$$

we would obtain $\mathcal{S} = \{1, 2, 4, 8, 9, 11, 15, 16, 18, 22, 23, 25\}$. Observe for every $i \in \mathcal{S}$ that $i \mod 7$ is in D_7.

In practice, only covers D_h with $|D_h| \in \Theta(\sqrt{h})$ are suitable. However, for the chosen D_h a function $\delta(i, j)$ must also be precomputed. For any integers i, j, $\delta(i, j)$ is the smallest integer $k \in 0..h-1$ such that $(i + k) \bmod h$ and $(j + k) \bmod h$ are both in D_h. A lookup table allows constant time evaluation of $\delta(i, j)$ — we omit the details here.

Algorithm BK consists of two main phases. The goal of the first phase is to compute a data structure $\mathrm{ISA}_{\boldsymbol{x}'}$ allowing the lexicographical rank of $i \in \mathcal{S}$, relative to the other members of $\mathcal{S}$, to be computed in constant time. To this end, BK first h-sorts $\mathcal{S}$ using MKQS (or alternative) and then assigns each suffix its h-rank in the resulting h-ordering. For our example the h-ranks are:

$i \in \mathcal{S}$	1	2	4	8	9	11	15	16	18
h-rank	3	6	4	3	6	4	2	5	1

These ranks are then used to construct a new string $\boldsymbol{x}'$ (compare to $\boldsymbol{x}'$ for Algorithm KS) as follows

$$\begin{array}{rc} i \in \mathcal{S} & \begin{array}{ccccccccc} 1 & 8 & 15 & 2 & 9 & 16 & 4 & 11 & 18 \end{array} \\ \boldsymbol{x}' = & (3\ 3\ 2)\ (6\ 6\ 5)\ (4\ 4\ 1) \end{array}$$

The structure of $\boldsymbol{x}'$ is deceptively simple. The h-ranks, r_i, appear in $|D_h|$ groups in $\boldsymbol{x}'$ (indicated above with ()) according to i modulo h. Then, within each group, ranks r_i are sorted in ascending order according to i. Because of this structure in $\boldsymbol{x}'$, its inverse suffix array, $\mathrm{ISA}_{\boldsymbol{x}'}$, can be used to obtain the rank of any $i \in \mathcal{S}$ in constant time. To compute ISA', BK makes use of Algorithm LS as an auxiliary routine (recall that LS computes both the ISA and the SA). Although LS is probably the best choice, any SACA suitable for bounded integer alphabets can be used.

With $\mathrm{ISA}_{\boldsymbol{x}'}$ computed, construction of $\mathrm{SA}_{\boldsymbol{x}}$ can begin in earnest. All suffixes are h-ordered using a string sorting algorithm, such as MKQS, to arrive at SA_h. The sorting of non-singleton h-groups which remain is then completed with a comparison based sorting algorithm using $\mathrm{ISA}_{\boldsymbol{x}'}[i + \delta(i, j)]$ and $\mathrm{ISA}_{\boldsymbol{x}'}[j + \delta(i, j)]$ as keys when comparing suffixes i and j.

In [BK03] it is shown that by choosing $h = \log_2 n$ an overall worst case running time of $O(n \log n)$ is achieved. Another attractive feature of BK is its small working space – less then $6n$ bytes – made possible by the small size of $\mathcal{S}$ relative to $\boldsymbol{x}$ and by use of inplace string sorting.

Finally, we remark that the ideas of Algorithm BK can be used to ensure any of the induced copying algorithms described in this section execute in $O(n \log n)$ worst case time.

Maniscalco [M05]

Algorithm M differs from the other algorithms in this section in that it directly computes $\mathrm{ISA}_{\boldsymbol{x}}$ and then transforms it into $\mathrm{SA}_{\boldsymbol{x}}$ inplace. At the time of writing, Algorithm M is published as C++ code on the Internet [M05], the details of which are examined in [P05].

At the heart of Algorithm M is an efficient bucket sorting regime. Most of the work is done in what is eventually $\mathrm{ISA}_{\boldsymbol{x}}$, with extra space required for a few stacks. The bucket sorting begins by linking together suffixes that are 2-equal, to form *chains* of suffixes. For example, the string

	0	1	2	3	4	5	6	7
$\boldsymbol{x}$	a	a	a	b	a	b	a	a	\$

would result in the creation of the following chains

7	6,1,0	4,2	5,3
a\$	aa	ab	ba

We define an h-chain in the same way as an h-group – that is, suffixes i and j are in the same h-chain iff they are h-equal. Thus, the chains above are all 2-chains, and the chain for a\$ is a singleton.

The space allocated for the ISA provides a way to efficiently manage chains. Instead of storing the chains explicitly as above, Algorithm M computes the equivalent array

	0	1	2	3	4	5	6	7
$\boldsymbol{x}$	a	a	a	b	a	b	a	a	\$
ISA	$\perp$	0	$\perp$	$\perp$	2	3	1	$\perp$	

in which $\mathrm{ISA}[i]$ is the largest $j < i$ such that $x[j..j+1] = x[i..i+1]$ or $\perp$ if no such j exists. In our example, the chain of all the suffixes prefixed with aa contains suffixes 6, 1 and 0 and so we have $\mathrm{ISA}[6] = 1$, $\mathrm{ISA}[1] = 0$ and $\mathrm{ISA}[0] = \perp$, marking the end of the chain. Observe that chains are singly linked, and are only traversable right-to-left. We keep track of h-chains to be processed by storing a stack of integer pairs (s, h), where s is the start of the chain (its rightmost index), and h is the length of the common prefix. Chains always appear on the stack in ascending lexicographical order, according to $x[s..s+h-1]$. Thus for our example, initially $(7, 2)$ for chain a\$ is atop the stack, and $(5, 2)$ for chain ba at the bottom.

Chains are popped from the stack and progressively refined by looking at further pairs of characters. So long as we process the chains in lexicographical order, when we pop a singleton chain, the suffix contained has been differentiated from all others and can be assigned the next lexicographic rank. Elements in the ISA which are ranks are differentiated from elements in chains by setting the sign bit, that is, if $\mathrm{ISA}[i] < 0$, then the rank for suffix i is $-\mathrm{ISA}[i]$. The evolution of the ISA of our example string subsequent sorting rounds proceed as follows.

```
formInitialChains()
repeat
  (h, ℓ) ← chainStack.pop()
  if ISA[h] = ⊥ then
    ISA[h] ← nextRank()
  else
    while h ≠ ⊥ do
      sym ← getSymbol(h + ℓ)
      updateSubChain(sym,h)
      h ← ISA[h]
    sortAndPushSubChains()
until chainstack is empty
```

Figure 7: Bucket sorting of Algorithm M

	0	1	2	3	4	5	6	7	
$\boldsymbol{x}$	a	a	a	b	a	b	a	a	\$
ISA	⊥	0	⊥	⊥	2	3	1	⊥	*Initial chains* $(7,2)_{a\$}(6,2)_{aa}(4,2)_{ab}(5,2)_{ba}$
ISA	⊥	0	⊥	⊥	1	2	1	-1	*Pop* $(7,2)_{a\$}$ *and assign rank*
ISA	⊥	⊥	⊥	⊥	1	2	⊥		*Split chain* $(6,2)_{aa}$ *into* $(6,4)_{aa\$}(0,4)_{aaab}(1,4)_{aaba}$
ISA	-3	-4	⊥	⊥	1	2	-2		*Pop* $(6,4)_{aa\$}(0,4)_{aaab}(1,4)_{aaba}$*, assign ranks*
ISA			⊥	⊥	⊥	2			*Split chain* $(4,2)_{ab}$ *into* $(4,4)_{abaa}(2,4)_{abab}$
ISA			-6	⊥	-5	2			*Pop* $(4,4)_{abaa}(2,4)_{abab}$*, assign ranks*
ISA				⊥		⊥			*Split chain* $(5,2)_{ba}$ *into* $(5,4)_{baa\$}(3,4)_{baba}$
ISA				-8		-7			*Pop* $(5,4)_{baa\$}(3,4)_{baba}$*, assign ranks*
ISA_x	3	4	6	8	5	7	2	1	*Completed Inverse Suffix Array*

When the value in a column becomes negative, the suffix has been assigned its (negated) rank and is effectively sorted. We reiterate here that when a chain is split, the resulting subchains must be placed on the stack in lexicographical order for the subsequent assignment of ranks to singletons to be correct. This is illustrated in the example above when the chain for aa is split, and the next chain processed is the singleton chain for $aa\$$. An algorithm embodying these ideas is listed in Figure 7.

Algorithm M adds two powerful heuristics to the string sorting algorithm described in Figure 7. We discuss only the first (and more important) of these heuristics here and refer the reader to [M05, P05] for details of the second.

The processing of chains in lexicographical order allows for the possibility to use previously assigned ranks as sort keys for some of the suffixes in a chain. To elucidate this idea we first need to make some observations about the way chains are processed.

When processing an h-chain, suffixes can be classified into three types: suffix i is of type X if the rank for suffix $i + h - 1$ is known, and is of type Y if the rank for suffix $i + h$ is known. If i is not of type X or type Y, then it is of type Z. Any suffix can be classified to its type in constant time by virtue of the fact we are building the ISA (we inspect $\text{ISA}[i + h - 1]$ or $\text{ISA}[i + h]$ and a checked sign bit indicates a rank). Now consider the following observation: lexicographically, type X suffixes are smaller than type Y suffixes, which in turn are smaller than type Z suffixes.

To use this observation, when we refine a chain, we place only type Z suffixes into subchains and place type X and type Y suffixes to one side. Now, the order of the m suffixes of type X suffixes can be determined via a comparison based sort, using for suffix i the rank of suffix $i+h-1$ as the sort key. Once sorted, the type X suffixes can be assigned the next m ranks by virtue of the fact that chains are being processed in lexicographical order. Type Y suffixes are treated similarly, using the rank of $j+h$ as the sort key for suffix j. Maniscalco refers to the sorting of suffixes in this way as *induction sorting* [2].

Loosely speaking, as the number of assigned ranks increases, the probability that a suffix can be sorted using the rank of another also increases. In fact, every chain of suffixes with prefix $\alpha_1\alpha_2$ such that $\alpha_2 < \alpha_1$ will be sorted entirely in this way. Clearly, induction sorting will lead to a significant reduction in work for many texts.

One could consider the induction sorting of Algorithm M an extension of the ideas in Algorithm IT. As noted above, suffixes in a 2-chain with common prefix $\alpha_1\alpha_2$ and $\alpha_1 > \alpha_2$ are sorted entirely by induction (like the type A suffixes of Algorithm IT. However the lexicographical processing of suffixes in Algorithm M means this property can be applied to suffixes at deeper levels of sorting (when $h > 2$).

The complexity of Algorithm M is likely to be $O(n^2 \log n)$ in the worst case, though on average it is usually as fast as Algorithm MF. By carefully using the space in the ISA, and converting it to the SA inplace, it also achieves a small memory footprint — rarely requiring more than n bytes of additional working space.

4 Experimental Results

To gauge the performance of the SACAs in practice we measured their runtimes and peak memory usage for a selection of files from the Canterbury corpus[3] and from the corpus compiled by Manzini[4] and Ferragina [MF04]. Details of all files tested are given in Table 2.

We implemented Algorithm IT as described in [IT99] and Algorithm KS with heuristics described in [PST05]. The implementation of Algorithm KA tested was that of [LP04]. Implementations of all other algorithms were obtained either online or by request to respective authors. For completeness we also tested a tuned suffix tree implementation [K99]. Algorithm MF was run with default parameters and Algorithm SS with parameter h=7 for genomic data and h=3 otherwise, as per testing in [SS05]. Algorithm BK used parameter h=32, as per [BK03].

All tests were conducted on a 2.8 GHz Intel Pentium 4 processor with 2Gb main memory. The operating system was RedHat Linux Fedora Core 1 (Yarrow) running kernel 2.4.23. The compiler was g++ (gcc version 3.3.2) executed with the -O3 option. Running times, shown in Table 3, are the average of four runs and do not include time spent reading input files. Times were recorded with the standard unix `time` function. Memory usage, shown in Table 4, was recorded with the `memusage` command available with most Linux distributions.

Results are summarized in Figure 8. Algorithm MF is the fastest algorithm on

[2] In fact, we can sort the type X and Y suffixes in the same sort call by using as a key for a type X suffix i the rank of $i+h-1$ and for a type Y suffix the *negated* rank of $i+h$.

[3] `http://www.cosc.canterbury.ac.nz/corpus/`

[4] `http://www.mfn.unipmn.it/~manzini/lightweight/corpus/`

Table 2: Description of the data set used for testing. LCP refers to the Longest Common Prefix amongst all suffixes in the string.

String	Mean LCP	Max LCP	Size (bytes)	σ	Description
E.coli	17	2,815	4,638,690	4	*Escherichia coli* genome
chr22.dna	1,979	199,999	34,553,758	4	Human chromosome 22
bible	14	551	4,047,392	63	King James bible
world192	23	559	2,473,400	94	CIA world fact book
sprot34	89	7,373	109,617,186	66	SwissProt database
rfc	93	3,445	116,421,901	120	Concatenated IETF RFC files
howto	267	70,720	39,422,105	197	Linux Howto files
reuters	282	26,597	114,711,151	93	Reuters news in XML format
jdk13c	679	37,334	69,728,899	113	JDK 1.3 documentation
etext99	1,108	286,352	105,277,340	146	Texts from Gutenberg project

average, narrowly shading algorithms M and SS. These three algorithms (MF,M,SS) outperform the next fastest algorithm, LS, by roughly a factor of 2. Note that for file jdk13c it is the suffix tree which is fastest — leaving room for at least some improvement in the SACAs.

When testing algorithm M, we observed that the final step of transforming the ISA into the SA constituted 20-30% of the overall runtime. For some applications though (most notably the BWT [BW94]), this transformation is not required, making M significantly faster than MF – see experiments in [P05].

The speed of MF and M is particularly impressive given their small working memory $5.01n$ and $5.49n$ bytes on average respectively. The lightweight nature, of these algorithms separates them from SS which requires slightly more than $9n$ bytes on average. We also remark that while Algorithm BK is not amongst the fastest algorithms tested the ideas in it are important because they could be used to guarantee acceptable worst case behavior of algorithms MF and M, without adversely affecting the speed or space usage of those algorithms.

Times in Table 3 for Algorithm SS versus Algorithm MF seem to run contrary to results published in [SS05], however our experiment is different. In [SS05], files were bounded to at most 50,000,000 characters, making many test files shorter than their original form. We suspect the full length files are harder for Algorithm SS to sort.

The large variation in performance of Algorithm KS can be attributed to the occasional ineffectiveness of heuristics described in [PST05]. Of interest also is the general poor performance of the recursive algorithms KS, KA and KJP. These algorithms have superior asymptotic behaviour, but for many files run several times slower than the other algorithms and often consume more memory than the suffix tree (KJP in particular). Memory profiling reveals that the recursive algorithms suffer form very poor cache behaviour, which largely nullifies their asymptotic advantage. These results leave open the question: is there a practically fast $\Theta(n)$ time suffix array construction algorithm which is also lightweight?

Table 3: CPU time (seconds) on test data. Minimum is shown in bold for each string.

	E.coli	chr22	bible	world	sprot	rfc	howto	reuters	jdk13c	etext
M	**2**	20	2	**1**	90	89	25	**99**	60	**75**
SS	**2**	25	2	**1**	99	93	22	133	64	92
MF	**2**	**16**	2	**1**	**74**	**65**	**18**	147	82	76
IT	**2**	416	**1**	**1**	125	108	38	278	286	331
S	3	29	2	**1**	126	110	37	258	217	290
BK	4	40	3	2	200	171	43	280	152	141
LS	4	35	3	2	144	154	40	183	105	146
KA	6	47	5	3	183	179	63	185	98	202
KS	5	57	4	2	306	288	55	377	204	219
KJP	4	31	4	3	183	189	61	192	102	179
Tree	6	51	5	3	183	193	80	141	**52**	226

Table 4: Peak Memory Usage (Mbs)

	E.coli	chr22	bible	world	sprot	rfc	howto	reuters	jdk13c	etext
M	32	205	29	13	547	599	197	572	357	542
SS	40	297	36	24	942	1,006	368	988	604	915
MF	22	165	19	12	524	557	188	548	333	503
IT	22	165	19	12	523	555	188	547	332	502
S	22	165	19	12	523	555	188	547	332	502
BK	26	194	23	14	614	652	221	643	391	590
LS	35	264	31	19	836	888	301	875	532	803
KA	58	429	50	31	1,359	1,443	526	1,422	864	1,406
KS	43	334	37	23	1,279	1,230	389	1,434	870	1,071
KJP	58	427	58	36	1,574	1,673	571	1,645	1,000	1,509
Tree	74	541	54	32	1,421	1,554	526	1,444	931	1,405

References

[AKO04] Mohamed Ibrahim Abouelhoda, Stefan Kurtz & Enno Ohlebusch, **Replacing suffix trees with enhanced suffix arrays**, *J. Discrete Algs. 2* (2004) 53–86.

[BK03] Stefan Burkhardt & Juha Kärkkäinen, **Fast lightweight suffix array construction and checking**, *Proc. 14th Ann. Symp. Combinatorial Pattern Matching*, LNCS 2676, Springer-Verlag (2003) 55-69.

[BM93] Jon L. Bentley & M. Douglas McIlroy, **Engineering a sort function**, *Software — Practice & Experience 23–11* (1993) 1249–1265.

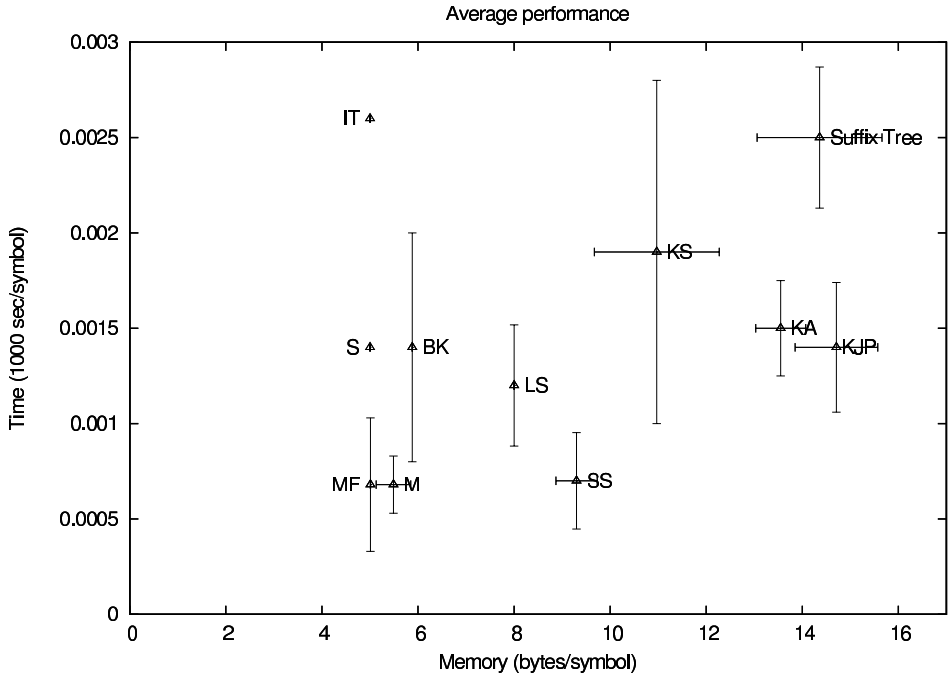


Figure 8: Resource requirements of the algorithms averaged over the test corpus. Error bars are one standard deviation. Abscissa error bars for algorithms MF, S, IT, BK and LS are insignificantly small. Ordinate error bars for algorithms S and IT are not shown to improve presentation (sd 0.009 and 0.0036 respectively).

[BS97] Jon L. Bentley & Robert Sedgewick, **Fast algorithms for sorting and searching strings**, Proc. ACM-SIAM Symp. Discrete Algs. (1997) 360–369.

[BW94] Michael Burrows & David J. Wheeler, *A Block-Sorting Lossless Data Compression Algorithm*, Research Report 124, Digital Equipment Corporation (1994) 18 pp.

[CF02] Andreas Crauser & Paolo Ferragina, **A theoretical and experimental study on the construction of suffix arrays in external memory**, *Algorithmica 32–1* (2002) 1–35.

[F97] Martin Farach, **Optimal suffix tree construction with large alphabets**, *Proc. 38th IEEE Symp. Found. Comp. Sci.* (1997) 137–143.

[FG99] Paolo Ferragina & Roberto Grossi, **The string B-tree: a new data structure for string search in external memory and its applications**, *J. Assoc. Comput. Mach. 46–2* (1999) 236–280.

[GGV04] Roberto Grossi, Ankur Gupta & Jeffrey Scott Vitter, **When indexing equals compression: experiments with compressing suffix arrays and applications**, *Proc. 15th ACM-SIAM Symp. Discrete Algs.* (2004) 636–645.

[H04] Michael Hart, *Project Gutenberg*: `http://www.gutenberg.net`

[HSS03] Wing-Kai Hon, Kunihiko Sadakane & Wing-Kin Sung, **Breaking a time-and-space barrier in constructing full-text indices**, *Proc. 44th IEEE Symp. Found. Comp. Sci.* (2003) 251–260.

[IT99] Hideo Itoh & Hozumi Tanaka, **An efficient method for in memory construction of suffix arrays**, *Proc. IEEE Symp. String Process. & Inform. Retrieval* (1999) 81–88.

[K99] Stefan Kurtz, Reducing the space requirement of suffix trees, *Software — Practice & Experience 29–13* (1999) 1149–1171.

[KA03] Pang Ko & Srinivas Aluru, **Space Efficient Linear Time Construction of Suffix Arrays**, *Proc. 14th Ann. Symp. Combinatorial Pattern Matching*, LNCS 2676, Springer-Verlag (2003) 200–210.

[KJP04] Dong Kyue Kim, Junha Jo & Heejin Park, **A fast algorithm for constructing suffix arrays for fixed-size alphabets**, *Proc. Workshop on Experimental Algorithms*, LNCS 3059, Springer-Verlag (2004) 301–314.

[KLAAP01] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa & Kunsoo Park, **Linear-time longest-common-prefix computation in suffix arrays and its applications**, *Proc. 12th Ann. Symp. Combinatorial Pattern Matching*, LNCS 2089, Springer-Verlag (2001) 181-192.

[KMR72] Richard M. Karp, Raymond E. Miller & Arnold L. Rosenberg, **Rapid identification of repeated patterns in strings, trees and arrays**, *Fourth Annual ACM Symp. Theory of Comput.* (1972) 125–136.

[KS03] Juha Kärkkäinen & Peter Sanders, **Simple Linear Work Suffix Array Construction**, *Proc. 30th Internat. Colloq. Automata, Languages & Programming*, LNCS 2719, Springer-Verlag (2003) 943–955.

[KSPP03] Dong Kyue Kim, Jeong Seop Sim, Heejin Park & Kunsoo Park, **Linear-time Construction of Suffix Arrays**, *Proc. 14th Ann. Symp. Combinatorial Pattern Matching*, LNCS 2676, Springer-Verlag (2003) 186–199.

[LP04] Sunglim Lee & Kunsoo Park, Efficient implementations of suffix array construction algorithms, *Proc. 15th Australasian Workshop on Combinatorial Algs.* , Seok-Hee Hong (ed.) (2004) 64–72.

[LS99] N. Jesper Larsson & Kunihiko Sadakane, *Faster Suffix Sorting*, Technical Report LU-CS-TR:99–214, Lund University (1999) 20 pp.

[M97] M. Douglas McIlroy, *ssort.c*:
`http://cm.bell-labs.com/cm/cs/who/doug/source.html`

[M99] J. Ian Munro, **Succinct data structures**, *Proc. Workshop on Data Structures* (1999) 3–7.

[M04] Giovanni Manzini, **Two space saving tricks for linear time LCP array computation**, *Proc. Scandinavian Workshop on Algorithm Theory* (2004) 372–383.

[M05] Michael Maniscalco, *MSufSort*: `http://www.michael-maniscalco.com/`

[MBM93] Peter M. McIlroy, Keith Bostic & M. Douglas McIlroy, **Engineering radix sort**, *Computing Systems 6–1* (1993) 5–27.

[MF04] Giovanni Manzini & Paolo Ferragina, **Engineering a lightweight suffix array construction algorithm**, *Algorithmica 40* (2004) 33–50.

[MM90] Udi Manber & Gene W. Myers, **Suffix Arrays: A new method for on-line string searches**, *Proc. First ACM-SIAM Symp. Discrete Algs.* (1990) 319–327.

[MM93] Udi Manber & Gene W. Myers, **Suffix Arrays: A new method for on-line string searches**, *SIAM J. Comput. 22* (1993) 935–948.

[N05] Jeong Chae Na, **Linear-time construction of compressed suffix arrays using $O(n \log n)$-bit working space for large alphabets**, *Proc. 16th Ann. Symp. Combinatorial Pattern Matching*, LNCS 3537, Springer-Verlag (2005) 57–67.

[P05] Simon J. Puglisi, *Exposition and analysis of a suffix sorting algorithm*, Technical Report CAS-05-02-WS, Department of Computing and Software, McMaster University (2005) 19 pp.

[PST05] Simon J. Puglisi, Bill F. Smyth & Andrew Turpin, **The performance of linear time suffix sorting algorithms**, *Proc. Data Compression Conf.* (2005) 358–367.

[S98] Kunihiko Sadakane, **A fast algorithm for making suffix arrays and for Burrows-Wheeler transformation**, *Proc. Data Compression Conf.* (1998) 129–138.

[S00] Julian Seward, **On the performance of BWT sorting algorithms**, *Proc. Data Compression Conf.* (2000) 173–182.

[S03] Bill F. Smyth, *Computing Patterns in Strings*, Pearson Addison-Wesley (2003) 423 pp.

[SKPP03] Jeong Seop Sim, Dong Kyue Kim, Heejin Park & Kunsoo Park, **Linear-time search in suffix arrays**, *Proc. 14th Australasian Workshop on Combinatorial Algs.* (2003) 139–146.

[SZ04] Ranjan Sinha & Justin Zobel, **Cache-conscious sorting of large sets of strings with dynamic tries** *ACM J. Experimental Algs. 9* (2004) 1–31.

[SS05] Klaus-Bernd Schürmann & Jens Stoye, **An incomplex algorithm for fast suffix array construction**, *Proc. 7th Workshop on Algorithm Engineering & Experiments*

Asynchronous Pattern Matching – Metrics (Extended Abstract)*

Amihood Amir

Bar-Ilan University and Georgia Tech
Department of Computer Science
52900 Ramat-Gan
ISRAEL

e-mail: amir@cs.biu.ac.il

Abstract. Traditional Approximate Pattern Matching (e.g. Hamming distance errors, edit distance errors) assumes that various types of errors may occur to the data, but an implicit assumption is that the order of the data remains unchanged.

Over the years, some applications identified types of "errors" were the data remains correct but its order is compromised. The earliest example is the "swap" error motivated by a common typing error. Other widely known examples such as transpositions, reversals and interchanges are motivated by biology.

We propose that it is time to formally split the concept of "errors in data" and "errors in address" since they present different algorithmic challenges solved by different techniques. The "errors in address" model, which we call asynchronous pattern matching, since the data does not arrive in a synchronous sequential manner, is rich in problems not addresses hitherto.

We will consider some reasonable metrics for asynchronous pattern matching, such as the number of inversions, or the number of generalized swaps, and show some efficient algorithms for these problems. As expected, the techniques needed to solve the problems are not taken from the standard pattern matching "toolkit".

1 Motivation

Historically, approximate pattern matching grappled with the challenge of coping with errors in the data. The traditional *Hamming distance* problem assumes that some elements in the pattern are erroneous, and one seeks the text locations where this number of errors is small enough [17, 14, 4], or efficiently calculating the Hamming distance at every text location [1, 16, 4]. The *edit distance* problem adds to the assumption that some elements of the text are deleted, or that noise is added at some text locations [18, 11]. Indexing and dictionary matching under these errors has also been considered [15, 12, 21, 10].

Implicit in all these problems is the assumption that there may indeed be errors in the **content** of the data, but the **order** of the data is inviolate. Data may be

*Partially supported by NSF grant CCR-01-04494 and ISF grant 82/01.

lost or noise may appear, but the relative position of the symbols is unchanged. Data **does not move around**. Even when *don't cares* were added [13], when non-standard models were considered[6, 20, 2] the order of the data was assumed to be ironclad.

Nevertheless, some non-conforming problems have been gnawing at the walls of this assumption. The *swap* error, motivated by the common typing error where two adjacent symbols are exchanged [19, 3], does not assume error in the content of the data, but rather, in the order. The data content is, in fact, assumed to be correct. Recently, the advent of computational biology has added more problems of order error to our repertoire. In evolution, one envisions a whole piece of genome to "detach" and "reconnect" in a different location, or two pieces of genome to "exchange" places. These phenomena, of course, are assumed to take place simultaneously with traditional data content errors, however, their nature is **rearrangement** of the data, rather than corruption of its contents.

It turns out that the overall problem of adding these new rearrangement operators to the content changing operators is extremely difficult. Thus more simplified problems were considered in the literature. The rearrangement operators were isolated and handled separately. Reversals [7], transpositions [5], and block interchanges [9] were explored. The edit distance problem under these new operations is still too difficult, therefore the *sorting permutation* version of these problems was researched.

This research direction led to interesting paths. First, the tools and techniques used were different from the traditional pattern matching tools. The results also seem more varied. The *sorting by reversal problem* is $\mathcal{NP}$-hard [8]. It is still open whether the *sorting by transposition problem* can be efficiently solved deterministically. Christie [9] gives an $O(n^2)$ algorithm for the *sorting by block interchange problem.*

In this paper, for the first time, we explicitely identify and formalize this different pattern matching paradigm, that of **errors in the order** rather than error in the content of the data. The advantages in formalizing this paradigm are:

1. Identifying the types of problems and techniques required, rather than than re-inventing ad-hoc solutions.

2. Understanding the theoretical underpinnings of the problem.

3. Generalizing to other possible rearrangements and possibly providing more general solutions.

One of the immediate understandings from a formal model definition of errors in order, is that one needs to consider appropriate *distance measures*. The error in content measures are not necessarily meaningful in these circumstances. We consider some generic error distances, such as minimum L_1 and L_2 distance on the address of the data. We also illustrate the fact that more specific distance measures are necessary for specific applications.

The main contributions of this research are: We give a formal framework of *rearrangement operators* and the distance measure they define. We also provide efficient algorithms for several natural operators and distance measures. It is exciting to point out that some techniques we use are totally new to pattern matching. This reinforces the realization that this new model is needed, as well as gives hopes to new research directions and paths in the field of pattern matching.

2 The New Model

We begin by an illustration of different applications requiring different rearrangement operators.

An Example. At the Formula-one races, cars and their designated drivers queue behind the start line at a precise, predetermined order. Suppose that the cars arrive at random order, and then have to rearrange into order. There is only a single passing lane, so that at any given time only one pair of cars can swap locations. What is the minimal number of swaps necessary in order to complete the rearrangement? Suppose that instead of reshuffling cars, the cars stay in place, and the drivers exchange cars. To do so, the drivers meet mid way and swap keys. In this case, multiple swaps can occur in parallel. What is the minimal number of parallel steps necessary in order to get all the drivers in order? Customarily, race cars and drivers are divided into groups. Suppose that the initial queuing order determines the ordering by group, not by specific car and driver. What is the minimum number of steps for the rearrangement in this case (sequential and parallel)?

Our new model considers how to efficiently answer these and similar questions.

Rearrangement Systems and Distances. Consider a set A and let x and y be two n-tuples over A. We wish to formally define the process of converting x to y through a sequence of *rearrangement* operations. A *rearrangement operator* π is a function $\pi : [1..n] \to [1..n]$, with the intuitive meaning being that for each i, π moves the element currently at location i to location $\pi(i)$. Let Π be a set of rearrangement operators, and let $w : \Pi \to R^+$ be a cost function, associating a non-negative cost with each operator. We call the pair (Π, w) a *rearrangement system*. Consider two vectors $x, y \in A^n$ and a rearrangement system $\mathcal{R} = (\Pi, w)$, we define the distance from x to y under $\mathcal{R}$. Let $s = (\pi_1, \pi_2, \ldots, \pi_k)$ be a sequence of rearrangement operators from Π, and let $\pi_s = \pi_1 \circ \pi_2 \circ \cdots \circ \pi_k$ be the composition of the π_j's. We say that s *converts* x *into* y if for any $i \in [1..n]$, $x_i = y_{\pi_s(i)}$. That is, y is obtained from x by moving elements according to the designated sequence of rearrangement operations. The *cost* of the sequence s is the sum of costs of the different operators in s, $w(s) = \sum_{j=1}^{k} w(\pi_j)$. The distance from x to y under $\mathcal{R}$ is defined as:

$$d_R(x, y) = \min\{w(s) | s \text{ converts } x \text{ to } y \}$$

If there is no sequence that converts x to y then the distance is ∞.

We extend the definition to tuples of different lengths. In this case, we define the distance between the two vectors to be the minimum distance between the shorter of the two and the closest contiguous subsequence of the longer.

We consider several natural rearrangement systems and the resulting distances. For these systems we provide efficient algorithms to compute the distances.

The Swaps Distance. We first consider the set of rearrangement operators were in each operation the location of exactly two entries can be swapped (as in the car rearrangement example above). The cost of each swap is 1. We call the resulting distance the *swaps distance*. We prove:

Theorem 1. *For tuples x and y of sizes m and n respectively ($m \leq n$) where all entries of x are distinct the swaps distance can be computed in time $O(m(n-m+1))$.*

The Parallel-Swaps Distance. Next we consider the case were in each rearrangement operation multiple pairs can be swapped, but any element can participate in at most one swap per operation (as in the drivers swap example above). Formally, this corresponds to the set of all permutation with cycles of length at most 2. The cost of each such permutation is 1. We call the resulting distance the *parallel swaps distance*, denoted by $d_{p\text{-}swap}(\cdot,\cdot)$. We prove:

Theorem 2. *For any two tuples x and y, either $d_{p\text{-}swap}(x,y) = \infty$ or $d_{p\text{-}swap}(x,y) \leq 2$.*

This means that for any two tuples x and y that are identical as multi-sets, it is possible to convert one to the other using only two parallel steps of swap operations! We also prove:

Theorem 3. *For tuples x and y of sizes m and n respectively ($m \leq n$) with k distinct entries in x, the parallel swaps distance can be computed deterministically in time that is the minimum of $O(k^2 n \log m)$ and $O(m(n-m+1))$.*

and,

Theorem 4. *For tuples x and y of sizes m and n respectively ($m \leq n$), the parallel swaps distance can be computed randomly in expected time that is the minimum of $O(n \log m)$ and $O(m(n-m+1))$.*

The L_1 Rearrangement Distance. Consider the set of rearrangement operations where in each operation exactly one element is moved. The element can be moved to any other location, and the cost of the operation is the distance the element is moved. We call this the L_1 *Rearrangement System* and the resulting distance the L_1 *Rearrangement Distance.* We prove:

Theorem 5. *For tuples x and y of sizes m and n respectively ($m \leq n$), the L_1 Rearrangement Distance can be computed in time $O(m(n-m+1))$. If all entries of x are distinct, then the distance can be computed in time that is the minimum of $O(n \log\log m)$ and $O(m(n-m+1))$*

The L_2 Rearrangement Distance. Consider the same set of operations as in the L_1 Rearrangement System, only that the cost of an operation is the square of the distance. We call this the the L_2 *Rearrangement System*, and the resulting distance the L_2 *Rearrangement Distance.* We prove:

Theorem 6. *For tuples x and y of sizes m and n respectively ($m \leq n$), the L_2 Rearrangement Distance can be computed in time that is the minimum of $O(n \log m)$ and $O(m(n-m+1))$*

References

[1] K. Abrahamson. Generalized string matching. *SIAM J. Comp.*, 16(6):1039–1051, 1987.

[2] A. Amir, A. Aumann, R. Cole, M. Lewenstein, and E. Porat. Function matching: Algorithms, applications, and a lower bound. In *Proc. 30th ICALP*, pages 929–942, 2003.

[3] A. Amir, R. Cole, R. Hariharan, M. Lewenstein, and E. Porat. Overlap matching. *Information and Computation*, 181(1):57–74, 2003.

[4] A. Amir, M. Lewenstein, and E. Porat. Faster algorithms for string matching with k mismatches. *J. Algorithms*, 2004.

[5] V. Bafna and P.A. Pevzner. Sorting by transpositions. *SIAM J. on Discrete Mathematics*, 11:221–240, 1998.

[6] B. S. Baker. A theory of parameterized pattern matching: algorithms and applications. In *Proc. 25th Annual ACM Symposium on the Theory of Computation*, pages 71–80, 1993.

[7] P. Berman and S. Hannenhalli. Fast sorting by reversal. In D.S. Hirschberg and E.W. Myers, editors, *Proc. 8th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 1075 of *LNCS*, pages 168–185. Springer, 1996.

[8] A. Carpara. Sorting by reversals is difficult. In *Proc. 1st Annual Intl. Conf. on Research in Computational Biology (RECOMB)*, pages 75–83. ACM Press, 1997.

[9] D. A. Christie. Sorting by block-interchanges. *Information Processing Letters*, 60:165–169, 1996.

[10] R. Cole, L. Gottlieb, and M. Lewenstein. Dictionary matching and indexing with errors and don't cares. In *Proc. 36th annual ACM Symposium on the Theory of Computing (STOC)*, pages 91–100. ACM Press, 2004.

[11] R. Cole and R. Hariharan. Approximate string matching: A faster simpler algorithm. In *Proc. 9th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 463–472, 1998.

[12] P. Ferragina and R. Grossi. Fast incremental text editing. *Proc. 7th ACM-SIAM Symposium on Discrete Algorithms*, pages 531–540, 1995.

[13] M.J. Fischer and M.S. Paterson. String matching and other products. *Complexity of Computation, R.M. Karp (editor), SIAM-AMS Proceedings*, 7:113–125, 1974.

[14] Z. Galil and R. Giancarlo. Improved string matching with k mismatches. *SIGACT News*, 17(4):52–54, 1986.

[15] M. Gu, M. Farach, and R. Beigel. An efficient algorithm for dynamic text indexing. *Proc. 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 697–704, 1994.

[16] H. Karloff. Fast algorithms for approximately counting mismatches. *Information Processing Letters*, 48(2):53–60, 1993.

[17] G. M. Landau and U. Vishkin. Efficient string matching with k mismatches. *Theoretical Computer Science*, 43:239–249, 1986.

[18] V. I. Levenshtein. Binary codes capable of correcting, deletions, insertions and reversals. *Soviet Phys. Dokl.*, 10:707–710, 1966.

[19] R. Lowrance and R. A. Wagner. An extension of the string-to-string correction problem. *J. of the ACM*, pages 177–183, 1975.

[20] S. Muthukrishnan and H. Ramesh. String matching under a general matching relation. *Information and Computation*, 122(1):140–148, 1995.

[21] S. C. Sahinalp and U. Vishkin. Efficient approximate and dynamic matching of patterns using a labeling paradigm. *Proc. 37th FOCS*, pages 320–328, 1996.

From Suffix Trees to Suffix Vectors

Élise Prieur and Thierry Lecroq

ABISS
University of Rouen
76281 Mont-Saint-Aignan, France

e-mail: {elise.prieur,thierry.lecroq}@univ-rouen.fr

Abstract. We present a first formal setting for suffix vectors that are space economical alternative data structures to suffix trees. We give two linear algorithms for converting a suffix tree into a suffix vector and conversely. We enrich suffix vectors with formulas for counting the number of occurrences of repeated substrings. We also propose an alternative implementation for suffix vectors that should outperform the existing one.

Keywords: Suffix tree, suffix vector, repeats.

1 Introduction

A suffix vector is an alternative data structure to a suffix tree. A suffix vector, for a string y, can store, in a reduced space, the same information as in a suffix tree of y. Suffix vectors have been introduced by Monostori [4, 5, 6] in order to detect plagiarism. The suffix vector of the string y consists in a succession of boxes located at some positions on the string y. These boxes are equivalent to the nodes of the suffix tree of y. Monostori gave an "on-line" linear construction algorithm of an extended suffix vector and a linear algorithm to compact a vector.

We are the first to give a formal setting for suffix vectors. To do that we describe two linear algorithms to convert a suffix tree into a suffix vector and conversely. We also supply suffix vectors with counters of the number of occurrences of repeated substrings for a given length. From practical experiences, we propose an alternative physical implementation for the suffix vectors that should outperform the one proposed by Monostori. This article is organized as follows: Section 2 introduces the different notations and quickly recalls suffix trees; Section 3 introduces suffix vectors; Section 4 shows the conversion from a suffix tree to a suffix vector; Section 5 gives the conversion from a suffix vector to a suffix tree; Section 6 presents a method for counting the number of occurrences of repeated substrings in a string; Section 7 discusses the suffix vector implementation and finally Section 8 gives our conclusions and perspectives.

2 Notations

Let A be a finite alphabet. Throughout the article we will consider a string $y \in A^*$ of length n: $y = y[0..n-1]$. We append to y the symbol \$ as a terminator which does not belong to A. From now on, y is a string of length $n+1$ finishing with \$.

The suffix tree $\mathcal{T}(y)$ of y is a linear size index structure that contains all the suffixes of y from the empty one to y itself. It can be constructed by considering the suffix trie of y (tree containing all the suffixes of y which edges are labeled by exactly one letter) where all internal nodes with only one child are removed and where remaining successive edge labels are concatenated. The leaves of the suffix tree contain the starting position of the suffix they represent.

The total length of all the suffixes of y can be quadratic, the linear size of the suffix tree is thus obtained by representing edge labels by pairs (*position*, *length*) referencing factors $y[position..position + length - 1]$ of y. The terminator \$ ensures that no suffix of y is an internal factor of y and thus $\mathcal{T}(y)$ has exactly $n + 1$ leaves. Each internal node has at least two children, leading to at most n internal nodes and thus a linear number of nodes overall. This also gives a linear number of edges. Each edge requires a constant space. Altogether the suffix tree $\mathcal{T}(y)$ of y can be stored in linear size. Figure 1(a) presents $\mathcal{T}(\texttt{aatttatttatta\$})$.

There exist several linear time suffix tree construction algorithms [3, 7, 1] that extensively use the notion of suffix links.

Each node p of the tree is identified with the substring obtained by concatenating the labels on the unique path from the root to the node p. We represent the existence of the edge from node p to node q with label (i, ℓ) by $\delta(p, (i, \ell)) = q$. We also consider $\textsc{Target}(p, a)$ which can be defined as $\delta(p, (i, \ell))$ for $y[i] = a$ and $\ell \geqslant 1$. For $a \in A$ and $u \in A^*$, if au is a node of $\mathcal{T}(y)$ then $s(au) = u$ is the suffix link of the node au.

For instance, in Figure 1:

- node 7 in the tree is identified with $\texttt{atttatt}$,
- the edge going from node 3 to node 7 is $\delta(\texttt{att}, (4, 4)) = \texttt{atttatt}$,
- and $\textsc{Target}(\texttt{att}, \texttt{t}) = \delta(\texttt{att}, (4, 4)) = \texttt{atttatt}$.

The right position of the first occurrence of the string u in y is denoted by $rpos(u, y)$, for instance $rpos(\texttt{att}, \texttt{aatttatttatta\$}) = 3$.

3 Suffix vectors

3.1 Extended suffix vectors

The suffix vector $\mathcal{V}(y)$ of y is a linear representation of the suffix tree $\mathcal{T}(y)$ consisting in a succession of boxes. These boxes contain the same information as the nodes of the tree, so that all the repeated substrings of y are represented in $\mathcal{V}(y)$.

Monostori did not give any formal definition of the suffix vectors, he only gave a linear time construction algorithm. We will now give a description of the suffix vectors.

There is a correspondence between the lines of the boxes of the suffix vector and the nodes of the suffix tree. Let B_j be the box of the suffix vector at position j of the string y. The box B_j is considered as an array with k lines and 3 columns. The first column contains the depth of the node, the second one contains the natural edge. The natural edge of a node p in a box B_j is the position of the box containing the node q such that $\textsc{Target}(p, y[j + 1]) = q$.

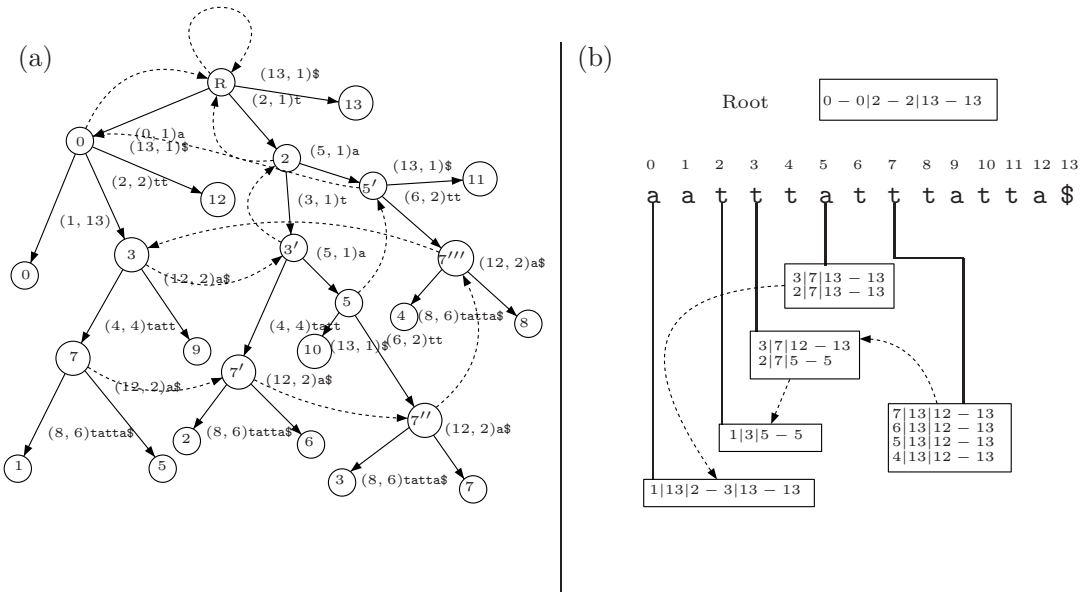


Figure 1: (a) Suffix tree of the string `aatttatttatta$`. The edges are labelled by pairs (*position*, *length*) and the substrings represented by the pairs. The label of the edge from node 0 to leaf 0 corresponds to the substring `atttatttatta$`. (b) Suffix vector of `aatttatttatta$`. Suffix links are represented by dashed arrows.

The third column of a box B_j contains the edge lists L. Each edge $L[g]$ of L is stored as a pair (b, e). We use $b = L[g].b$ and $e = L[g].e$, b is the beginning of the edge (the position of the first character) and e the end of the edge (the position of the box containing the target node). So a box is characterized by: $B[h, 0] = depth, B[h, 1] = ne, B[h, 2] = L$ for each $0 \leqslant h \leqslant k - 1$.

Inside a box, there are implicit suffix links from node represented by depth d to node represented by depth $d-1$. The depth of the deepest node is also stored in each box. Monostori pointed out in [4] that the depths in a box are continuous.

The root of the suffix tree is represented by a specific box in the suffix vector.

Example

In the box B_3 in the vector of Figure 1(b), the first line indicates that there exists a node representing a substring u of length 3 with $rpos(u, y) = 3$, so $u =$ `att`. Its natural edge is 7, this means that there is an edge from u such that $\textsc{Target}(u, y[4])$ is a node in B_7. The length of this edge is 4 (7-3), so this is the node of depth 7 in B_7 which recognizes `atttatt`.

The list of edges $B_3[1, 2]$ contains $12 - 13$. This means that there is one edge (different of the natural edge) going out from this node, its label begins at position 12 and ends at position 13. The end position is equal to the length of y, so this edge leads to a leaf.

1

We now present an example of utilization of a suffix vector. Let y be the string `aatttatttatta$` and x be the string `tatt`. We use the suffix vector of y (Figure 1(b)) to know whether x is a substring of y. In the edge list of the root, there is an edge labeled $(2, 2)$ and $y[2] =$ `t`, so we follow it and go to the box at position 2. This box

has only one line. As $y[3] \neq \mathtt{a}$, we do not follow the natural edge. The only edge in $B_2[0,2]$ begins at position 5, $y[5] = \mathtt{a}$ so we can follow it. It leads to the box B_5. As we have already read the prefix $\mathtt{ta}$ of x, we consider the line representing the node of depth 2. Since $y[6] = \mathtt{t}$, we follow the natural edge which leads to the box at position 7, so its length is 2. As $y[6..7] = \mathtt{tt}$, we have found one occurrence of x in y.

3.2 Compact suffix vectors

We introduce here the notion of compact suffix vector. A suffix vector can be compacted when, for lines h_1 and h_2 of the box at position j, the edge list of line h_1 is included in the edge list of line h_2: $B_j[h_1, 2] \subseteq B_j[h_2, 2]$. In this case, we just need to store the list of the line h_2 and create a link between the two lists. These boxes are called reduced boxes. They contain the number of nodes. To compact a suffix vector, Monostori established three rules (see [4]). These three compaction rules are:

Rule A the node with depth $d-1$ has the same number of edges as the node with depth d and these are the same edges. In this case we simply set their first edge pointers to the same position.

Rule B the node with depth $d-1$ has the same edges as the node with depth d plus some extra edges. In this case, the list of edges of the node with depth $d-1$ contains its own edges and a pointer to the list of edges of the node with depth d.

Rule C the node with depth $d-1$ has different edges to the node with depth d. In this case, all the edges must be represented in a separate list.

These rules are illustrated in Figure 2. Monostori gave a linear time algorithm for compacting an extended suffix vector.

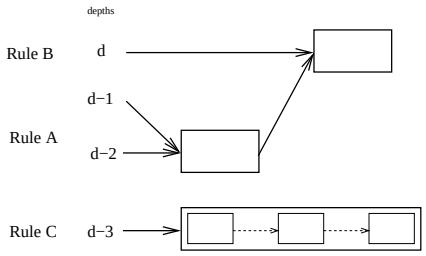


Figure 2: Representation of the compaction rules

Example

In the vector of Figure 1, we note that, in the boxes at positions 5 and 7, only the depths differ between the lines. So these boxes could be compacted storing only the first line and the number of lines. The result of the compaction of this suffix vector is shown Figure 3.

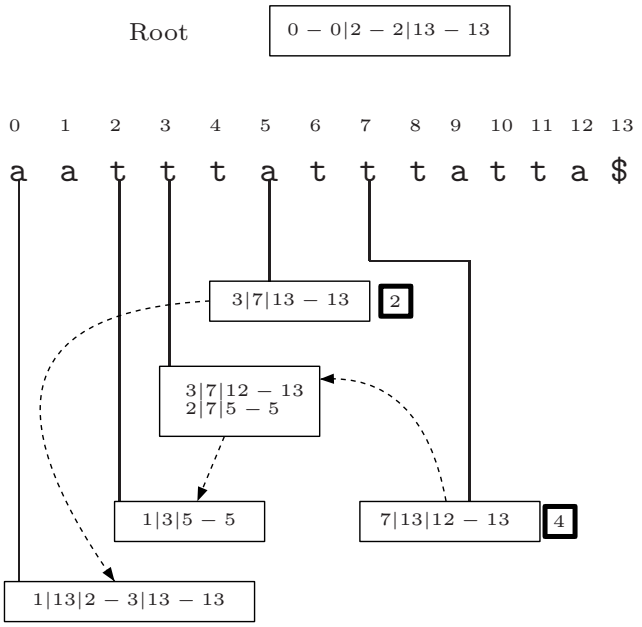


Figure 3: Compact suffix vector of the string `aatttatttatta$`.

4 Converting a suffix tree into a suffix vector

4.1 Method

We first outline the principle of the conversion of a suffix tree into an extended suffix vector by giving some propositions. The first one establishes the correspondence between an internal node in the suffix tree and a line in a box in the extended suffix vector.

Proposition 4.1. *Let p be an internal node of $\mathcal{T}(y)$ such that $y[j-d+1..j]$ is the first occurrence of the substring represented by p. This implies that there exists a box at position j in the suffix vector $\mathcal{V}(y)$ and a line h in B_j such that $B_j[h,0] = d$.*

Proof

Let $u \in \mathcal{T}(y)$ be a node of the suffix tree of the string y, u is a substring of y. When u is a node it means that it has at least two occurrences in y, this implies that u is represented in $\mathcal{V}(y)$ since all the repeated substrings of y are represented in $\mathcal{V}(y)$.

We denote by $j = rpos(u, y)$ the right position of the first occurrence of u in y. So, there exists a box B_j at position j in the vector. In this box, there exists a line h such that $B_j[h,0] = |u|$. If $u = y[j-d+1..j]$, we have $B_j[h,0] = d$.

Line h is such that among all the substrings $w \in \mathcal{T}(y)$ such that $rpos(w, y) = j$, u is the $(h+1)$-th longest one. ∎

Example

In the tree of Figure 1, node 5 can be identified with the substring $u =$ `tta` of y for which $rpos(u, y) = 5$. This node verifies Proposition 4.1 since the first line of the box B_5 in the vector represents a node of depth 3 ($B_5[0,0] = 3$ and $|$`tta`$| = 3$).

The next proposition establishes the correspondence between an edge in the suffix tree and either the natural edge or one edge in an edge list of the suffix vector.

Proposition 4.2. *Let (i, ℓ) be an edge of $\mathcal{T}(y)$ such that $\delta(p, (i, \ell)) = q$ where p and q are nodes of the suffix tree. Node p is such that $y[j-d+1..j]$ is the first occurrence of the substring represented by p. Two cases can arise:*

1. *(i, ℓ) is the natural edge of p ($y[i] = y[j+1]$), then $B_j[h, 1] = j + \ell$;*
2. *(i, ℓ) is an edge such that $y[i] \neq y[j+1]$ then there exists a pair (b, e) in $B_j[h, 2]$ such that $b = i$ and $e = i + \ell - 1$.*

Proof

Node p satisfies Proposition 4.1.

1. The natural edge:

   Considering the substring $u = y[j-d+1..j]$ the edge beginning with the letter $y[j+1]$ gives that there exists a node at position $rpos(\textsc{Target}(u, y[j+1]), y) = j + \ell$. The number $j + \ell$ is either the length of y (so the edge leads to a leaf) or the right position of the substring $u \cdot y[j+1..j+\ell]$. In the latter case, after Proposition 4.1, there exists a box at position $j + \ell$. Thus, $j + \ell$ is obtained following the natural edge of the node at line h in B_j. This implies that $B_j[h, 1] = j + \ell$.

2. The others edges:

   Considering $\textsc{Target}(u, y[i]) = q$ such that $rpos(\textsc{Target}(u, y[i]), y) = i + \ell - 1$ with $y[i] \neq y[j+1]$. The number $i + \ell - 1$ is either the length of y (so the edge leads to a leaf) or the right position of the substring $u \cdot y[i..i+\ell-1]$. In the latter case, after Proposition 4.1, there exists a box at position $i + \ell - 1$. Then, in box B_j there exists an edge $L[g] \in B_j[h, 2]$ such that $L[g].e = rpos(\textsc{Target}(u, y[i]), y) = i + \ell - 1$ and $L[g].b = L[g].e - |\textsc{Target}(u, y[i])| + 1 + |u| = i$.

■

Example

In the tree of Figure 1(a), there is an edge going out from node 5 beginning with $y[6] = \texttt{t}$ and labeled by $(6, 2)$. This node can be identified with the substring $u = \texttt{tta}$ of y for which $rpos(u, y) = 5$. It is represented by the first line of B_5. We have $rpos(\textsc{Target}(\texttt{tta}, y[6]), y) = rpos(\texttt{ttatt}, y) = 7$ so $B_5[0, 1] = 7$. This is the natural edge, this verifies Proposition 4.2 since $B_5[0, 1] = 5 + 2 = j + \ell$.

Node 5 in the suffix tree possesses only one edge beginning by a caracter in $A \setminus \{y[6]\}$ labeled by $(13, 1)$, $rpos(\textsc{Target}(\texttt{tta}, \$), y) = 13$ (this is a leaf) and $|\textsc{Target}(\texttt{tta}, \$)| - 1 - |\texttt{tta}| = 0$. The second part of Proposition 4.2 holds because in the box we have: $B_5[1, 2] = L$ such that L has one element defined by $L[0].e = 13$ and $L[0].b = 13$.

In the next proposition, the special case of the root is processed.

```
TREE2VECT(T(y))
    ▷ R is the root of the tree T(y)
 1  ADDROOT(R, V(y))
 2  for each child node p of R such that p is not a leaf do
 3  |  PUSH(S, p)
 4  while not STACK-EMPTY(S) do
 5  |  p ← POP(S)
 6  |  ADDNODE(p, V(y))
 7  |  for each child node q of p such that q is not a leaf do
 8  |  |  PUSH(S, q)
 9  return V(y)
```

Figure 4: Algorithm converting a suffix tree into a suffix vector.

Proposition 4.3. *Each edge (i, ℓ) going out from the root of the tree is represented by the pair $(i, i + \ell - 1)$ in the edge list of the specific box of the root of the suffix vector.*

Proof

Similar to the proof of Proposition 4.1.

■

The next two propositions show the correspondence for the suffix links.

Proposition 4.4 (Theorem 5.1 of [4]). *Let $s(u) = v$ be a suffix link in $\mathcal{T}(y)$ such that $rpos(u, y) = rpos(v, y)$ then u and v are represented in the same box of $\mathcal{V}(y)$.*

Proposition 4.5. *Let $s(u) = v$ be a suffix link in $\mathcal{T}(y)$ such that $i = rpos(u, y) \neq rpos(v, y) = j$ then $s(B_i) = B_j$.*

Proof

The suffix links are only defined from internal nodes to internal nodes. After Proposition 4.1, node u is represented in the box at position i and node v in the box at position j.

■

4.2 Algorithm

We now describe the algorithm to get a suffix vector from a suffix tree. For each node p of $\mathcal{T}(y)$, we need to know the value $rpos(p, y)$. This can be computed if each node p stores its length $|p|$ and the position of the first occurrence of p which corresponds to the number of the smallest leaf in the subtree rooted at p. This algorithm is based on a depth-first search of the suffix tree. It ensures the visit of all the nodes of the suffix tree. We use a stack S to visit the nodes (see Figure 4).

First, the algorithm processes the root because, in the vector, the root is not represented as the other nodes. The function ADDROOT, called line 1 in Figure 4, adds all the edges going out from the root of the tree in the root list of the suffix vector. It is described Figure 5.

```
ADDROOT(R, V(y))
    ▷ LR is the list representing the root of V(y)
 1  LR ← ∅
 2  for each edge (i, ℓ) going out from R do
 3  |  INSERT((i, i + ℓ − 1), LR)
```

Figure 5: Algorithm adding the root of a suffix tree into a suffix vector.

```
ADDNODE(p, V(y))
 1  j ← rpos(p, y)
 2  if ∄B_j then
 3  |  CREATE(B_j)
 4  |  h ← 0
 5  else h ← k
    |  ▷ k is the number of lines in B_j
 6  |  k ← k + 1
 7  B_j[h, 0] ← |p|
 8  for each edge (i, ℓ) going out from p do
 9  |  if y[i] = y[j + 1] then
    |  |  ▷ this is the natural edge
10  |  |  B_j[h, 1] ← j + ℓ
11  |  else INSERT((i, i + ℓ − 1), B_j[h, 2])
12  if j ≠ rpos(s(p), y) then
13  |  s(B_j) ← B_{rpos(s(p),y)}
```

Figure 6: Algorithm adding a node of a suffix tree into a suffix vector.

Then, for each node p of the tree, we add its equivalent in the vector: we insert a line in a box at position $rpos(p, y)$ in the vector and if the box does not exist, we create it with the correct line. This function is detailed in Figure 6.

Theorem 4.1. *The algorithm* TREE2VECT($\mathcal{T}(y)$) *correctly computes* $\mathcal{V}(y)$ *in time* $O(|y|)$

Proof

The correctness of the algorithm comes from Propositions 4.1 to 4.5.

Each node and each edge of the suffix tree are processed only once. The operations per node and per edge take a constant time. Since the number of edges and nodes of the suffix tree is linear, the result on the running time follows.

■

5 Converting a suffix vector into a suffix tree

5.1 Method

We now show the conversion from an extended suffix vector to a suffix tree. The next proposition deals with the internal nodes.

Proposition 5.1. *Each line h of a box B_j in the suffix vector of y can be associated to an internal node of the suffix tree of y.*

Proof

Let u be the substring of y such that $u = y[j - B_j[h, 0] + 1..j]$. If there is a line h in a box B_j it means that $u \cdot y[j + 1..B_j[h, 1]]$ and $u \cdot y[L[0].e..L[0].b]$ are factors of y with $L[0] \in B_j[h, 2]$ and $y[j + 1] \neq y[L[0].e]$. This means that u has two occurrences in y followed by two different letters which implies that u represents an internal node in $\mathcal{T}(y)$. ■

Example

In the box B_5 of Figure 1(b), $B_5[0, 0] = 3$ indicates that the substring $u = y[5 - 3 + 1..5] = y[3..5]$ is represented in the first line of this box. This string is `tta`, it corresponds to node 5 in the suffix tree of y.

The three following propositions deal with the edges.

Proposition 5.2. *Each value $B_j[h, 1]$ of a line h of a box B_j in the suffix vector of y can be associated to an edge of the suffix tree of y.*

Proof

Let u be the substring of y such that $u = y[j - B_j[h, 0] + 1..j]$. There exists an edge in the tree such that $\delta(u, (j+1, B_j[h, 1]-j)) = y[j-B_j[h, 0]+1..B_j[h, 1]]$. Thus $y[j - B_j[h, 0] + 1..B_j[h, 1]]$ is in $\mathcal{T}(y)$, it can be an internal node or a leaf. ■

Example

In the box B_5 of Figure 1(b), the second column of the first line means that we can go to position 7 following an edge starting from position 6, this edge is $\delta(\texttt{tta}, (6, 2))$ in $\mathcal{T}(y)$.

Proposition 5.3. *Each pair (b, e) in a edge list of a line h of a box B_j in the suffix vector of y can be associated to an edge of the suffix tree of y.*

Proof

Let u be the substring of y such that $u = y[j - B_j[h, 0] + 1..j]$. There exists an edge in the tree such that $\delta(u, (b, e - b + 1)) = u \cdot y[e..b]$. Thus $u \cdot y[e..b]$ is in $\mathcal{T}(y)$, it can be an internal node or a leaf. ∎

Example

The third column of the first line of B_5 of Figure 1(b) has only one edge, $L[0].b = 13$ and $L[0].e = 13$ ($L[0].e = |y|$ means that this edge leads to a leaf). We have to verify that there exists an edge such that $\delta(\texttt{tta}, (L[0].b, L[0].e - L[0].b + 1)) = \delta(\texttt{tta}, (13, 1))$ in the tree. The node 5, which recognizes the same substring as the first line of B_5, has an edge labeled $(13, 1)$ going out to a leaf. We showed the equivalence between the node 5 in the tree and the first line of B_5 in the vector.

Proposition 5.4. *Each pair (b, e) in a edge list of the root the suffix vector of y can be associated to an edge of the suffix tree of y.*

Proof

Similar to Proposition 5.3. ∎

The next proposition deals with the leaves.

Proposition 5.5. *The leaves of the suffix tree $\mathcal{T}(y)$ can be retrieved from the suffix vector $\mathcal{V}(y)$.*

Proof

This is a direct consequence of Propositions 5.3 to 5.5 and the fact that there is exactly one edge leading to each leaf. ∎

The two following propositions deal with the suffix links.

Proposition 5.6 (Theorem 5.1 of [4]). *In a box B_j of k lines the suffix link of the node represented by the line h points to the node represented by the line $h + 1$ for $0 \leqslant h < k - 1$.*

Proposition 5.7. *In a box B_j of k lines the suffix link of the node represented by the line $k - 1$ points to $s(B_j)$.*

Proof

By construction. ∎

5.2 Algorithm

We give in this section an algorithm that computes a suffix tree from an extended suffix vector for a string y. It first processes the root box of the suffix vector and then processes sequentially each remaining box of the vector. For each box it sequentially processes each lines (see Figure 7).

```
VECT2TREE(V(y))
 1  R ← new node
 2  for each (b, e) in the edge list of the root box of V(y) do
 3  |  p ← new node at depth e − b + 1 at position e
 4  |  δ(R, (b, e − b + 1)) ← p
 5  for j ← 0 to n do
    |  ▷ k is the number of lines of the box B_j
    |  ▷ p is the node previously created at depth B_j[k − 1, 0] at position j
 6  |  q ← new node at depth B_j[k − 1, 0] + B_j[k − 1, 1] − j + 1 at position B_j[k − 1, 1]
 7  |  δ(p, (j + 1, B_j[h, 1] − j + 1)) ← q
 8  |  r ← p
 9  |  for each pair (b, e) ∈ B_j[k − 1, 2] do
10  |  |  q ← new node at depth B_j[k − 1, 0] + e − b + 1 at position e
11  |  |  δ(p, (b, e − b + 1)) ← q
12  |  s(p) ← s(B_j)
13  |  for h ← k − 2 to 0 do
    |  |  ▷ p is the node previously created at depth B_j[h, 0] at position j
14  |  |  q ← new node at depth B_j[h, 0] + B_j[h, 1] − j + 1 at position B_j[h, 1]
15  |  |  δ(p, (j + 1, B_j[h, 1] − j + 1)) ← q
16  |  |  for each pair (b, e) ∈ B_j[h, 2] do
17  |  |  |  q ← new node at depth B_j[h, 0] + e − b + 1 at position e
18  |  |  |  δ(p, (b, e − b + 1)) ← q
19  |  |  s(r) ← p
20  |  |  r ← p
21  return T(y)
```

Figure 7: Algorithm converting a suffix vector into a suffix tree.

Theorem 5.1. *The algorithm* VECT2TREE($\mathcal{V}(y)$) *correctly computes* $\mathcal{T}(y)$ *in time* $O(|y|)$.

Proof

The correctness of the algorithm comes from Proposition 5.1 to 5.6.

The algorithm processes each pairs of each lines of each boxes of the suffix vector which correspond to the edges and the nodes of the suffix tree whose quantity is linear. The only difficulty consists in retrieving a node at depth d for position j. This can be realized by storing the largest depth in each box. All the other operations take constant time. The result on the running time follows. ∎

6 Repeats

Before counting the number of repeats of the substrings of y, we explain some notions for the counting of the number of occurrences.

6.1 Counting the number of occurrences

As mentioned before, each line of a box of $\mathcal{V}(y)$ is associated to a node u in $\mathcal{T}(y)$ and thus to a substring of y. Let B_j be a box of $\mathcal{V}(y)$, let h be a line of B_j, the line h is

associated to the substring $u = y[j - B_j[h,0] + 1..j]$. Let $nbOcc(u)$ be the number of occurrences of the substring u. Let $nbL(t)$ be the number of leaves in the subtree rooted at the end of any edge t .

Then

$$nbL(t) = \begin{cases} 1 & \text{if } t = |y|; \\ nbOcc(v) & \text{otherwise} \end{cases}$$

where v is the node in the box B_t such that t is the end position of the edge going to node v.

We then deduce that

$$nbOcc(u) = nbL(ne) + \sum_{L[g] \in B_j[h,2]} nbL(L[g].e).$$

With this expression, it is easy to obtain a linear algorithm which adds the value $nbOcc(u)$ on each line of the vector. This algorithm visits the boxes of the suffix vector from right to left and completes the lines with $nbOcc(u)$.

6.2 Counting the number of repeats

The method described in this section allows to compute for each substring of y with a given length $lg < n$, its number of occurrences in y. Let $lpocc(lg)$ be the list of pairs $(rpos(u,y), nbOcc(u))$ for all substrings u of y of length lg.

The principle is to visit all the boxes of the suffix vector and for each line h in a box at position j such that $B_j[h,0] \geqslant lg$ to update $lpocc(lg)$ using $nbOcc$(u) where u is the substring represented by this line.

First, we test if the depth of the deepest node of the box we are visiting is larger than lg. In the contrary case, the visit of the box stops. For reduced boxes we only have to take into account the deepest node, whereas in the other boxes we have to process all the nodes whose depth is larger than lg. We now explain the two different cases.

Reduced box Let us assume that we are processing the reduced box B_j and $B_j[0,0] = d \geqslant lg$. This implies this line represents $u = y[j - d + 1..j]$. Let j' be a position such that $j - d + 1 \leqslant j' \leqslant d - k + 1$ and $|y[j'..j]| \geqslant lg$ (k is the number of lines represented in the box). For each possible j', let v be $y[j'..j' + lg - 1]$, either we add the pair $(rpos(v,y), nbOcc(u))$ in the list or we update $nbOcc(v)$ with $nbOcc(u)$ if v is already present in $lpocc$.

Extended box For each line h of the extended box B_j such that $B_j[h,0] = d \geqslant lg$. This implies this line represents $u = y[j - d + 1..j]$. Let v be the prefix of length lg of u, either we add the pair $(rpos(v,y), nbOcc(u))$ in the list or we update $nbOcc(v)$ with $nbOcc(u)$ if v is already present in $lpocc$.

After that, the list $lpocc(lg)$ gives the number of occurrences of repeated substring of y of length lg.

7 Implementation

7.1 Monostori's implementation

We now explain the representation used by Monostori to store compact suffix vectors (section 5.4 of [4]). Each box contains the following information:

Deepest node The deepest node value is usually small so Monostori proposed to store it in 1 or 4 bytes. The first bit is used to denote the number of bytes needed to store the value, so the deepest node value is represented with 7 or 31 bits.

Number of nodes In a box, we also need the number of nodes value which is smaller than the deepest node value. The number of nodes can fit into one byte when the deepest node value is stored into one byte. So, it is not necessary to use another bit to flag as for the depth.

Suffix link The next information stored in a box is the suffix link. If the number of nodes is equal to the depth of the deepest node, this means that the smallest depth in the box is 1. So the suffix link of the box is implicit to the root. In this representation, the suffix link is stored anyway because its first bit is used to indicate if the box is a reduced one and its second one is used to indicate if the values of the natural edges will need 1 or 4 bytes.

Natural edges Then the natural edges are stored in an array called array of next node pointers. We can save space by storing the length of an edge rather than the end position. If there is one of the lengths of the natural edges of the box which need to be stored in more than one byte, all of them are stored in 4 bytes.

Edges At last, we have to consider the representation of the edges. An edge is represented with its start position and its length. The first edge pointer of a node gives the memory address of the list of edges going out from this node. The first bit of a start position of an edge indicates if this edge leads to a leaf. In this case, the length of the edge is not stored. The next bit flags whether this is the last edge of the list. The third one is used to indicate the number of bytes (1 or 4) required to store the length of the edge.

7.2 Counting

Table 1 compares the space required by the suffix vector with the space required by the suffix tree implemented with Kurtz's method [2]. It is extracted from Table 5.1 in [4]. Here, we give the results for four files which are two English texts (`book2` and `bible`), one C program (`progc`) and one DNA sequence (`ecoli`). The results are given in bytes per symbol of the input sequences.

The measures done by Monostori show that its implementation of the suffix vectors is less efficient for DNA sequences than for large alphabets. Therefore we performed experiments on several DNA sequences. Tables 2 to 6 give the results for five of them:

- chromosome 4 of *S. cerevisiae* (of length 1,531,931);

Table 1: Comparison of space requirements of suffix vectors and suffix trees.

File name	File size (in bytes)	Bytes/symbol Compact Suffix Vector	Bytes/symbol Kurtz
book2	610,857	8.61	9.67
bible	4,047,393	8.53	7.27
progc	39,612	8.63	9.59
ecoli	4,638,691	12.51	12.56

- chromosome 3 of *E. coli* (of length 13,783,270);
- chromosome 5 of *E. coli* (of length 20,922,241);
- chromosome 2 of *A. thaliana* (of length 19,847,294);
- chromosome 4 of *A. thaliana* (of length 17,790,892).

For each sequence, we build its extended suffix vector and reported for each box:

- the number of nodes;
- the depth;
- the length of the natural edge minus 1 (since it is always at least equal to 1);

and for each edge list of each box:

- the next position;
- the difference between the next position and the position of the box minus 2 (since it is always at least equal to 2);
- the length of the edge minus 1 (since it is always at least equal to 1).

For all the values we counted the number of them that can fit between:

- 1 and 6 bits;
- 7 and 14 bits;
- 15 and 22 bits;
- 23 and 30 bits.

The idea is, instead of using only one flag bit and use 1 or 4 bytes for representing the different objects, to use two flag bits and 1, 2, 3 or 4 bytes for representing them. The tables clearly show that this approach will save a large number of bytes in all cases. Of course, storing the difference between the next position and the position of the box rather than the next position always enables to save storage space. The actual total gain is not yet completely measurable since, to keep a direct access to any node in a box, all the natural edges in a box are stored with the space necessary for the largest natural edge. We can now present an alternative implementation.

Table 2: Counts for chromosome 4 of *S. cerevisiae*. It contains 1,531,931 base pairs.

	1 – 6 bits	7 – 14 bits	15 – 22 bits	23 – 30 bits
Number of nodes	501,378	129		
Depth	875,852	13,924		
Natural edge	369,501	6497	513,778	
Next position	55	18,794	1,555,245	
Difference	2539	128,375	1,443,180	
Edge length	642,254	9143	922,697	

Table 3: Counts for chromosome 3 of *E. coli*. It contains 13,783,270 base pairs.

	1 – 6 bits	7 – 14 bits	15 – 22 bits	23 – 30 bits
Number of nodes	4,182,237	2283		
Depth	7,978,520	172,622		
Natural edge	3,697,663	32,220		4,421,259
Next position	2	18,527	4,529,723	9,302,660
Difference	36,315	439,978	11,548,079	1,826,540
Edge length	5,633,779	35,474		8,181,659

Table 4: Counts for chromosome 5 of *E. coli*. It contains 20,922,241 base pairs.

	1 – 6 bits	7 – 14 bits	15 – 22 bits	23 – 30 bits
Number of nodes	6,395,182	3224	4	
Depth	11,998,929	288,722	40,428	
Natural edge	5,456,836	121,698	5156	6,744,389
Next position	8	18,606	4,579,468	16,428,416
Difference	42,261	485,515	14,355,158	6,143,564
Edge length	8,583,613	111,606	2344	12,328,935

Table 5: Counts for chromosome 2 of *A. thaliana*. It contains 19,847,294 base pairs.

	1 – 6 bits	7 – 14 bits	15 – 22 bits	23 – 30 bits
Number of nodes	6,429,030	2192		
Depth	11,499,363	137,572	183,616	
Natural edge	5,353,725	32,671		6,434,155
Next position	61	18,470	4,630,471	15,624,486
Difference	37,770	426,015	14,124,810	5,691,893
Edge length	8,186,302	23,318		12,063,868

Table 6: Counts for chromosome 4 of *A. thaliana*. It contains 17,790,892 base pairs.

	1 – 6 bits	7 – 14 bits	15 – 22 bits	23 – 30 bits
Number of nodes	5,809,708	1869		
Depth	10,203,150	136,807	224,650	
Natural edge	4,767,148	33,801		5,763,658
Next position	61	18,713	4,578,990	13,529,684
Difference	26,541	474,155	13,353,092	4,273,660
Edge length	7,325,278	33,235		10,768,935

7.3 An alternative implementation

Here, we explain how to use the idea explained in section 7.2 to reduce the space. Each box contains the following information:

Deepest node Instead of storing in 1 or 4 bytes, we could store the depth of the deepest node in 1, 2, 3 or 4 bytes. This means that we have to use the two first bits to indicate how many bytes we need. So the deepest node value is stored in 6, 14, 22 or 30 bits.

Number of nodes As mentioned in Section 7.1, the number of bytes needed to store the number of nodes depends on the number of bytes of the deepest node value. Then, if we need 6, 14, 22 or 30 bits to store this depth, we could use respectively 1, 2, 3 or 4 bytes to store the number of nodes.

Suffix link Similar to Section 7.1.

Natural edges We can use two flag bits and then 1, 2, 3, or 4 bytes for all the values. We store the length of the natural edge minus 1 since it is always larger than 1.

Edges Instead of storing the start position of an edge, we could store the difference between the start position and the position of the box. For a box at position j and an edge starting at $L[g].b > j$, we store $L[g].b - j - 2$. We can use the same idea as for the deepest node value to store $L[g].b - j - 2$ and the length of $L[g] - 1$ using 1, 2, 3 or 4 bytes.

The main idea is to reduced the space required with Monstori's implementation for DNA sequences by storing the data with 2 or 3 bytes instead of 4 when it is possible. To do that we use 2 bits for the needed number of bytes. Tables 2 to 6 show that we can reduce the space in many cases.

8 Conclusions and perspectives

We presented a first formal setting for suffix vectors that are space economical alternative data structures to suffix trees. We gave two linear algorithms for converting a suffix tree into a suffix vector and conversely. We enriched suffix vectors with formulas for counting the number of occurrences of repeated substrings. We finally proposed an alternative implementation for suffix vectors that should outperform the one proposed by Monostori specially for small alphabets and large sequences.

In order to really take advantage of this implementation we are studying an "online" linear algorithm for directly building a compact suffix vector. This should allow to deal efficiently with huge sequences such as human chromosomes.

References

[1] M. Farach. Optimal suffix tree construction with large alphabets. In *Proceedings of the 38th IEEE Annual Symposium on Foundations of Computer Science*, pages 137–143, Miami Beach, FL, 1997.

[2] S. Kurtz. Reducing the space requirements of suffix trees. *Software Practice & Experience*, 29(13):1149–1171, 1999.

[3] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of Algorithms*, 23(2):262–272, 1976.

[4] K. Monostori. *Efficient Computational Approach to Identifying Overlapping Documents in Large Digital Collections*. PhD thesis, Monash University, 2002.

[5] K. Monostori, A. Zaslavsky, and H. Schmidt. Suffix vector: Space-and-time-efficient alternative to suffix trees. In *CRPITS '02: Proceedings of the 25th Australasian Computer Science Conference*, volume 4, pages 157–166, Melbourne, 2002. Australian Computer Society, Inc.

[6] K. Monostori, A. Zaslavsky, and I. Vajk. Suffix vector: A space-efficient suffix tree representation. In *Proceedings of the 12th International Symposium on Algorithms and Computation*, volume 2223 of *Lecture Notes in Computer Science*, pages 707–718, Christchurch, New Zealand, 2001. Springer Verlag.

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

Reconstructing a Suffix Array

F. Franek and W. F. Smyth

Algorithms Research Group
Department of Computing & Software
McMaster University
Hamilton, Ontario
Canada L8S 4K1

e-mail: franek@mcmaster.ca, smyth@mcmaster.ca

Abstract. For certain problems (for example, computing repetitions and repeats, data compression applications) it is not necessary that the suffixes of a string represented in a suffix tree or suffix array should occur in lexicographical order (lexorder). It thus becomes of interest to study possible alternate orderings of the suffixes in these data structures, that may be easier to construct or more efficient to use. In this paper we consider the "reconstruction" of a suffix array based on a given reordering of the alphabet, and we describe simple time- and space-efficient algorithms that accomplish it.

Keywords: suffix array, suffix tree, lexicographic order, alphabet, string

1 Introduction

We use a small example to introduce the main ideas. Consider the string

$$\begin{array}{rcccccc} & {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 4} & {\scriptstyle 5} & {\scriptstyle 6} \\ \boldsymbol{x} = & a & b & a & a & b & \$ \end{array}$$

whose suffix tree $T_{\boldsymbol{x}}$ is shown in Figure 1 (the conventional sentinel \$ is a lexicographically least letter introduced to ensure that every suffix of $\boldsymbol{x}$ is represented as a leaf node of $T_{\boldsymbol{x}}$).

Ignoring the sentinel suffix, a preorder traversal of $T_{\boldsymbol{x}}$ allows the suffix array of $\boldsymbol{x}$ to be read off in lexorder from the leaf nodes:

$$\begin{array}{rccccc} & {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 4} & {\scriptstyle 5} \\ \mathrm{pos} = & 3 & 4 & 1 & 5 & 2 \end{array} \tag{1}$$

with the lengths (***lcp***s) of the corresponding longest common prefixes (***LCP***s) read off from the internal nodes:

$$\mathrm{lcp} = 0 \quad 1 \quad 2 \quad 0 \quad 1. \tag{2}$$

Let us call the usual suffix array (for example, (1)) the ***lexicographical suffix array*** of $\boldsymbol{x}$ (LSA($\boldsymbol{x}$)), of course unique and well-defined for every string $\boldsymbol{x}$ on an ordered

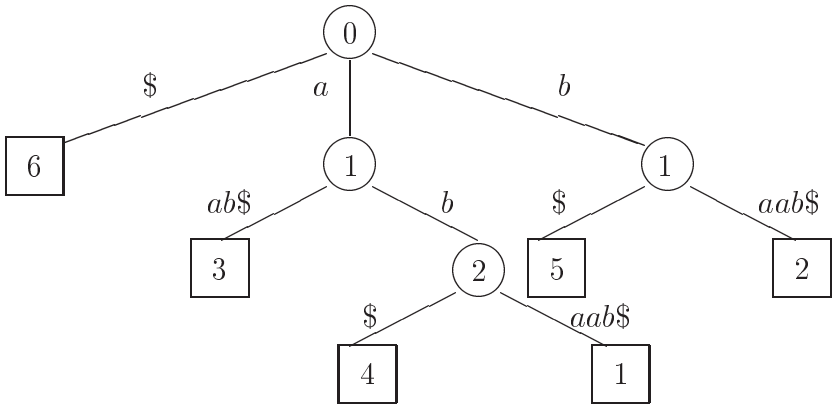


Figure 1: The suffix tree $T_{\boldsymbol{x}}$ of $\boldsymbol{x} = abaab$

alphabet. More generally, we may define a ***valid suffix array*** of $\boldsymbol{x}$ (VSA($\boldsymbol{x}$)) to be any reordering of LSA($\boldsymbol{x}$) that can be obtained by reordering the subtrees of $T_{\boldsymbol{x}}$, then reading off the terminal nodes (except the sentinel suffix) in a preorder traversal. For our example string $\boldsymbol{x} = abaab$, there are actually 16 VSAs of $\boldsymbol{x}$:

34152, 34125, 31452, 31425
41352, 41325, 14352, 14325
52341, 25341, 52314, 25314
52413, 25413, 52143, 25143

Observe that of course for a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ of length n, there are altogether $n!$ permutations of LSA($\boldsymbol{x}$); in our example 16 out of the $5! = 120$ permutations are actually VSAs. Note that if all the letters of $\boldsymbol{x}$ are distinct, then there will be $n!$ distinct VSAs of $\boldsymbol{x}$.

Finally we define a ***consistent suffix array*** of $\boldsymbol{x}$ (CSA($\boldsymbol{x}$)) to be a VSA that is determined by an ordering (reordering) of the alphabet. In our example, there are just two CSAs of $\boldsymbol{x}$:

34152 (for $\$ < a < b$) and 52413 (for $\$ < b < a$).

In this paper we present algorithms to compute the CSA($\boldsymbol{x}$) determined by a specified ordering of the alphabet, given the LSA. As explained below, we think of this research as an initial step in gaining an understanding of how to compute a CSA or a VSA directly, without intermediate steps that depend on the LSA or the suffix tree.

Suffix arrays (LSAs) were introduced in 1990 [MM90, MM93] as a more space-efficient alternative to suffix trees; at the same time an $O(n \log n)$ algorithm was described for their construction. In 1997 a linear-time suffix *tree* construction algorithm was proposed [F97], effective in the normal case that the alphabet is ***indexed*** — that is, essentially, a finite integer alphabet. In 2003, based on [F97], three different groups of researchers independently discovered linear-time recursive algorithms to compute the LSA [KA03, KS03, KSPP03], also on an indexed alphabet. It turns out, however, that, largely as a consequence of their recursive nature, these algorithms are generally slower in practice [PST05] than two other classes of LSA construction algorithms whose worst-case behaviour is supralinear: ***direct comparison*** algorithms

and ***prefix doubling*** algorithms. Direct comparison algorithms make use of a pointer copying method introduced in [BW94] to efficiently sort suffixes one letter at a time [IT99, S00, MF04]; although their worst-case time requirement can therefore be as much as $\Theta(n^2 \log n)$, they generally have low space requirements and execute very fast in practice. On the other hand, prefix doubling algorithms make use of a technique introduced in [KMR72] to roughly double the length of the suffixes sorted at each step [MM93, LS99, BK03]; their worst-case time bound is thus only $O(n \log n)$ and they also tend to execute quickly in practice. Of the algorithms tested in [PST05], that of Manzini & Ferragina [MF04] appears to hold an advantage, both in the use of space and time, over that of Burkhardt & Kärkkäinen [BK03] in second place, but algorithms more recently described [SS05, M05] may be still more efficient.

The curious (to us, at least) fact is that to date the most efficient known way to compute any VSA is to first compute the LSA($\boldsymbol{x}$). In [FSXH03] we have described algorithms that essentially compute VSAs, but these algorithms are not as fast as the best LSA construction algorithms, even though LSA construction in general requires fewer conditions to be satisfied. It seems to us that VSA construction should be in some sense easier than LSA construction, but as things stand the opposite is true.

In this paper we will suppose that LSA($\boldsymbol{x}$) has been computed for $\boldsymbol{x} = \boldsymbol{x}[1..n]$ based on an ordering $(\mathcal{A}, <)$ of the alphabet $\mathcal{A}$. Then we show how to construct

$$\mathrm{CSA}(\boldsymbol{x}) = \mathrm{LSA}'(\boldsymbol{x})$$

determined by a reordering $(\mathcal{A}, <')$ of $\mathcal{A}$. In Section 2 we describe two $\Theta(n)$-time algorithms to handle a special case that arose in a recent paper [FS05]: ***reverse lexorder***, where for any letters $\lambda, \mu \in \mathcal{A}$,

$$\lambda < \mu \iff \mu <' \lambda. \tag{3}$$

Section 3 presents an efficient algorithm for the general case: an arbitrary permutation of the order of the alphabet. Finally, Section 3 presents conclusions and outlines future work.

2 Reversing the Order of the Alphabet

As discussed in the Introduction, we assume that (3) holds, and we use LSA$[1..n]$ to denote the suffix array corresponding to $(\mathcal{A}, <)$, LSA$'[1..n]$ for the suffix array corresponding to $(\mathcal{A}, <')$. Recall that a ***border*** of a string $\boldsymbol{x}$ is any proper prefix of $\boldsymbol{x}$ that is also a suffix. We define the ***right border array*** $\beta = \beta[1..n]$ of $\boldsymbol{x}$ as follows: for every $i \in 1..n$, $\beta[i] = j \iff j$ is the length of the longest border of $\boldsymbol{x}[i..n]$. β can be computed in $\Theta(n)$ time and constant space using a straightforward variant of the standard (left) border array algorithm [S03, ex. 1.3.10]. Observe that $\beta[i]$ is the lcp not only of $\boldsymbol{u} = \boldsymbol{x}[n\beta[i]+1..n]$ and $\boldsymbol{v} = \boldsymbol{x}[i..n]$, but also of every suffix $\boldsymbol{w}$ of $\boldsymbol{x}$ that lies between $\boldsymbol{u}$ and $\boldsymbol{v}$ in lexorder.

For technical reasons to simplify the presentation of the following lemmas and algorithms, we modify slightly the array β: $\beta[i] \neq 0$ is not the length of the longest border of $\boldsymbol{x}[i..n]$, but the index of the suffix of $\boldsymbol{x}$ that is the longest border, i.e. $\beta[i] = j \neq 0$ if and only if $\boldsymbol{x}[j..n]$ is the longest border of $\boldsymbol{x}[i..n]$ (see Figure 2).

The algorithms for reverse lexorder are then a consequence of the following lemmas:

```
β[1] ← 0;
for i ← 1 to n−1 do
  if β[n−i+1] = 0 then
    c ← 0
  else
    c ← n+1−β[n−i+1]
  while c > 0 and x[n−i] ≠ x[n−c]
    if β[n−c+1] = 0 then
      c ← 0
    else
      c ← n+1−β[n−c+1]
  if x[n−i] = x[n−c] then
    β[n−i] ← n−c
  else
    β[n−i] ← 0
```

Figure 2: Computing $\beta[1..n]$ for input string $\boldsymbol{x}$[1..n]

Lemma 1. *Let* $j = \mathrm{LSA}[i]$ *for some* $i \in 1..n$.

(a) *If* $\beta[j] > 0$, *then* $\boldsymbol{x}\big[\beta[j]..n\big] <' \boldsymbol{x}[j..n]$;

(b) *otherwise, if* $\beta[j] = 0$, *then*

$$\boldsymbol{x}[j..n] <' \min_{1 \le h < i} \boldsymbol{x}\big[\mathrm{LSA}[h]..n\big]. \tag{4}$$

Proof If $\beta[j] > 0$, then $\boldsymbol{x}\big[\beta[j]..n\big]$ is a proper prefix of $\boldsymbol{x}[j..n]$, so that $\boldsymbol{x}\big[\beta[j]..n\big] <' \boldsymbol{x}[j..n]$. If $\beta[j] = 0$, then for every $h \in 1..i1$, there exists a least nonnegative integer $q_h \le \min\{nj+1, n\mathrm{LSA}[h]+1\}$ such that $\boldsymbol{x}\big[\mathrm{LSA}[h]+q_h\big] \ne \boldsymbol{x}[j+q_h]$. Thus by the definition of LSA, $\boldsymbol{x}\big[\mathrm{LSA}[h]{+}q_h\big] < \boldsymbol{x}[j{+}q_h]$, and so, by the definition of $<'$, $\boldsymbol{x}[j{+}q_h] <' \boldsymbol{x}\big[\mathrm{LSA}[h]{+}q_h\big]$. Hence (4) holds. □

Observe that every border of every suffix is represented by an entry in β and so will be covered by Lemma 1. Observe further that the quantities q_h introduced in the proof for $\beta[j] = 0$ are actually lcp values for each pair of suffixes $\boldsymbol{x}[j..n]$ and $\boldsymbol{x}\big[\mathrm{LSA}[h]..n\big]$.

Lemma 2. *Let* $j_1 = \mathrm{LSA}[i_1]$, $j_2 = \mathrm{LSA}[i_2]$, $1 \le i_1 < i_2 \le n$. *If* $\beta[j_1] = \beta[j_2] > 0$, *then*

$$\boldsymbol{x}[j_2..n] <' \boldsymbol{x}[j_1..n].$$

Proof Since $i_1 < i_2$, $\boldsymbol{x}[j_1..n] < \boldsymbol{x}[j_2..n]$; since neither of these strings can be a prefix of the other, the result follows. □

Figure 3 shows the simplest algorithm that computes LSA′. The algorithm illustrates the fundamental idea of the process in a clear and simple way. We suppose that the array β was computed in preprocessing, while the array NEXT[1..n] emulates a singly-linked list equivalent to LSA′ that is constructed as the input LSA is scanned from left to right (in increasing lexorder): we will consistently use the word

```
start ← LSA[1];
for i ← 2 to n do
  j ← LSA[i]
  if β[j] = 0 then
    — by Lemma 1 (b) j goes to start of list
    NEXT[j] ← start; start ← j
  else
    — by Lemmas 1 (a) & 2, insert j next to β[j]
    j' ← β[j]; temp ← NEXT[j']
    NEXT[j'] ← j; NEXT[j] ← temp
```

Figure 3: Algorithm 1 — Computing LSA′ for Reversed Alphabet

transform to refer to the computation of NEXT from LSA (and *vice versa*). We omit the straightforward **for** loop that transforms NEXT into LSA′.

```
 — transform LSA into NEXT
start ← LSA[1]
for i ← 1 to n−1 do
  NEXT[LSA[i]] ← LSA[i+1]
NEXT[LSA[n]] ← 0
compute β using memory storage of LSA
 — reorder NEXT
prev ← start; cur ← NEXT[prev]
while cur ≠ 0 do
  if β[cur] = 0 then  — cur goes to front
    NEXT[prev] ← NEXT[cur]; NEXT[cur] ← start
    start ← cur
  else  — cur goes next to β[cur]
    if NEXT[β[cur]] = cur then
      prev ← cur
    else
      NEXT[prev] ← NEXT[cur]; i ← NEXT[β[cur]];
      NEXT[β[cur]] ← cur; NEXT[cur] ← i
 — transform NEXT to LSA′ using memory storage of β
i ← 1; j ← start
for i ← 1 to n do
  LSA[i] ← j; j ← NEXT[j]
```

Figure 4: Algorithm 2 — Computing LSA′ for Reversed Alphabet

Algorithm 1 has the disadvantage of using $2|\boldsymbol{x}|$ words of working memory (the arrays β and NEXT) for the input string $\boldsymbol{x}$. Algorithm 2 (see Figure 4) is a bit more elaborate; however, it is based on the same principles as Algorithm 1 and uses only $|\boldsymbol{x}|$ words of working memory (for NEXT).
Thus

Theorem 1. *Given* LSA($\boldsymbol{x}$) *for a string* $\boldsymbol{x} = \boldsymbol{x}[1..n]$, *Algorithm 2 computes* LSA$'(\boldsymbol{x})$ *for a reversed alphabet in* $\Theta(n)$ *time using* n *words of working memory.*

Proof By induction. Clearly for $i = 1$ the entries in NEXT are in $<'$ order. Suppose that for arbitrary $i \in 1..n1$, the entries are in $<'$ order. By Lemmas 1 and 2, the entries must still be in $<'$ order after LSA$[i+1]$ has been processed. □

We note that essentially the same algorithm applies to a morphism $\sigma : \mathcal{A} \to \mathcal{B}$ from one ordered alphabet to another provided that for every distinct $\lambda, \mu \in \mathcal{A}$, $\lambda < \mu \iff \sigma(\mu) <' \sigma(\lambda)$.

3 Permuting the Order of the Alphabet

In this section we describe an algorithm to compute LSA$'(\boldsymbol{x})$ in the case of an arbitrary reordering $(\mathcal{A}, <')$ of the alphabet $\mathcal{A}$. Alternatively, we may think of this reordering as a permutation $\pi : \mathcal{A} \to \mathcal{A}$ where for every distinct $\lambda, \mu \in \mathcal{A}$, $\lambda < \mu \iff \pi(\lambda) <' \pi(\mu)$.

Essentially, our algorithm uses LSA($\boldsymbol{x}$) (in fact, as we shall see, any VSA($\boldsymbol{x}$) will do) to simulate a reordering of the subtrees of the suffix tree $T_{\boldsymbol{x}}$ that is determined by the reordering of the alphabet. In the simple example of Figure 1, the only possible reordering (since $|\mathcal{A}| = 2$, necessarily a reversal) would result from interchanging two paths in the subtree represented by a and b as well as in the subtree represented by $aab\$$ and ab, yielding LSA$'(\boldsymbol{x}) = 52413$.

It is instructive to consider the relationship between reversal and arbitrary reordering. In Lemma 1, if we suppose that $\beta[j] > 0$, it is true also in the general case that $\boldsymbol{x}\big[\beta[j]..n\big] <' \boldsymbol{x}[j..n]$; however, Lemmas 1 (b) and 2 no longer hold, since it is no longer possible to infer the order of $\boldsymbol{x}[j_1..n]$ and $\boldsymbol{x}[j_2..n]$ from the order in which they occur in LSA($\boldsymbol{x}$). In other words, the set of suffixes that have the same LCP $\boldsymbol{x}\big[\beta[j]..n\big]$ cannot simply be placed to the right of $\boldsymbol{x}\big[\beta[j]..n\big]$ — they must now be sorted in $<'$ order based on positions $\beta[j]+1, \beta[j]+2, \ldots$ in each suffix.

Similarly, in the case that $\beta[j] = 0$, (4) no longer holds: we must relocate suffixes by sorting in $<'$ order the ones that have the same LCP (occur in the same subtree of $T_{\boldsymbol{x}}$).

These comments imply that the array β is no longer useful in the general case, whereas the lcp array (for example, (2)) becomes critical. Fortunately, like β, the lcp array lcp$[1..n]$ can be computed in linear time, either from the LSA [KLAAP01] or as a byproduct of LSA construction: thus we assume throughout this section that it is available. In fact, as noted above, since in the general case the LSA ordering provides no information about the LSA$'$ ordering, the algorithm described in this section will work just as well using any VSA($\boldsymbol{x}$) together with its corresponding (permuted) lcp array.

Our algorithm reorders the suffixes of $\boldsymbol{x}$ beginning with those that share the greatest lcp values, thus equivalent to a traversal of the suffix tree $T_{\boldsymbol{x}}$ upwards from the deepest lcp nodes. We first outline the control structure that our algorithm uses to accomplish this traversal, then go on to describe the details of its implementation.

The input LSA($\boldsymbol{x}$) (VSA($\boldsymbol{x}$)) and its corresponding input lcp array LCP1 are being traversed from left to right in order to identify ***families***. In simple terms, a family is a set of nodes in the lis NEXTthat corresponds to a set of links to nodes that are

immediate children of an internal node of the corresponding suffix tree. These links can be permuted provided that the links in all subtrees have been already sorted. If the internal node that is the root of the subtree corresponds to lcp ℓ, we call the family an **ℓ-family**. A stack STACK for tracking families is maintained by the algorithm; if a value ℓ is on top of the stack, then an LCP[NEXT[ℓ]]-family starts at position NEXT[ℓ] (for technical reason we do not store the beginning of the family on the stack, but rather the previous node).

```
 — input: x - string
 — input: LSA- its suffix array
 — input: LCP1- lcp array for LSA
 — input: permutation p of the alphabet
NEXT[ ]  — auxiliary array
STACK  — stack for keeping track of families
Transform LSA to NEXT
Transform LCP1 to LCP using memory of LSA for LCP
use memory of LCP1 as memory for TAIL and initialize it
Initialize STACK and variables
while multipop()
  Identify and Extract a family (using STACK)
  Sort the family (using p)
  Flatten the family
  Verticalize the family
Sort the final 0-family
Flatten the final 0-family
Transform NEXT to LSA
Transform LCP to LCP1
 — output: LSA sorted according to p
 — output: LCP1 lcp of LSA
```

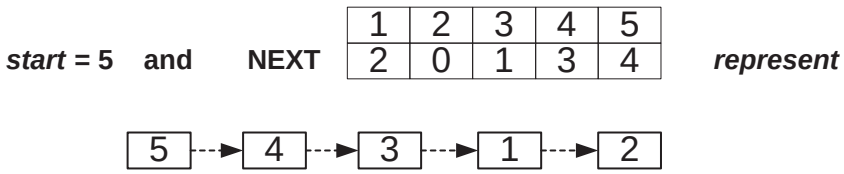
Figure 5: Outline of Algorithm 3 — General Reordering

The families are identified simply during the scan: as long as the values of LCP increase, they are pushed on the stack as they represent beginnings of families. A decreasing value indicates the end of the innermost family (i.e. the one on the top of the stack). After the family is sorted, it is "verticalized", so it is now represented as a single node in the family it is nested in and the scan can continue. One would expect to pop the stack once the innermost family is processed. However, the situation is a bit more complex, and thus ***multipop***() is employed to decide whether or not the stack should be popped. The control stucture of the algorithm is shown in Figure 5. The individual steps are described in detail below, making use of the following standard routines: $Push(s)$ pushes s on top of STACK, $Pop()$ pops STACK, $Top()$ obtains the value on the top of the stack STACK without popping it, $Top_1()$ obtains the value next to the top of STACK without popping it.

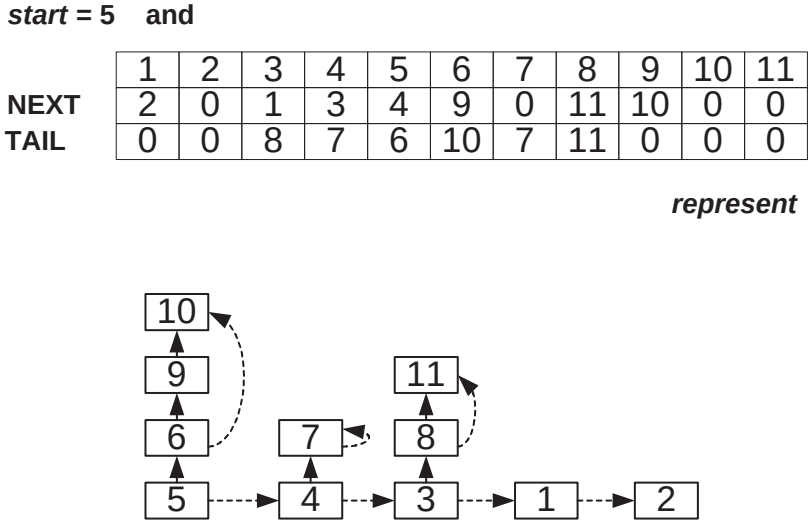
The data structures and variables

As shown in Figure 5, three arrays are used in addition to $\boldsymbol{x}$, two of them input, only one auxiliary. NEXT[1..n] emulates a singly-linked list of nodes, where each node

stores an integer value k representing the suffix $\boldsymbol{x}[k..n]$. A variable *start* marks the beginning of NEXT[ ]. For instance,



The array TAIL[n] represents the "verticalized" part of the list of nodes. For instance,



The end of each "vertical" tail is reachable in two steps: TAIL[TAIL[k]] is the very last member of the "vertical" tail starting at the node k. Other auxiliary variables used are: *cur* for a "pointer" to the current node in NEXT[ ], *prev* for a "pointer" to the previous node (if $prev = 0$, it means that $cur = start$). LE (*left end*) represents the node to which the head of a family is attached ($LE = 0$ means that the head of the family is *start*), RE (*right end*) represents the node to which the last member of a family will point to ($RE = 0$ means that the last member of a family is the last member of the NEXT list). Finally a variable *type* describes the type of family we are processing, i.e. the lcp of all memebers of the family.

Transform input LSA **to** NEXT

Traverse LSA and fill in the entries in NEXT:

$start \leftarrow$ LSA[1];
for $i \leftarrow 1$ **to** $n-1$ **do**
  NEXT[LSA[i]] $\leftarrow$ LSA[$i+1$]
NEXT[n] $\leftarrow 0$

Transform input LCP1 **to** LCP

Normally, LCP[i] represents the lcp of two neighbouring suffixes, $\boldsymbol{x}$[LSA[$i-1..n$]] and $\boldsymbol{x}$[LSA[$i..n$]]. But since during the sorting the mutual positions of suffixes can change, we modify the usual meaning to: LCP[i] represents the lcp of $\boldsymbol{x}$[LSA[$i..n$]] and its right neighbour. Thus, we traverse LCP1 and "shift" the values one position to the left. Since LSA is no longer needed, we use its memory for LCP:

```
LCP[start] ← LCP1[1];
for i ← 1 to n−1 do
  LCP[LSA[i]] ← LCP1[i+1]
LCP[n] ← 0
```

Initialize TAIL

Since LCP1 is no longer needed, we use its memory for TAIL. Since at the beginning we have no "vertical" tails, all entries must be initialized to 0:

```
for i ← 1 to n do
  TAIL[i] ← 0
```

Initialize STACK **and variables**

Start the traversal of NEXT and LCP. Keep traversing as long as LCP has value 0. Push on STACK *prev* of the first non-zero node.

```
prev ← 0; cur ← start
while LCP[i] = 0
  prev ← cur; cur ← NEXT[cur]
Push(prev)
type ← LCP[cur]
```

Identify and Extract a family

Note that we are now inside a loop (see Figure 5), and thus the use of the term **continue** means to transfer the flow of control to the top of the loop.

```
if LCP[cur] = type then
  prev ← cur; cur ← NEXT[cur]; continue
if LCP[cur] > type then  — a new family starts
  Push(prev)
  prev ← cur; cur ← NEXT[cur]; continue
if LCP[cur] < type then  — a family ends
  LE ← Top(); RE ← NEXT[cur]; NEXT[cur] ← 0
```

Thus we have just identified an innermost family of type LCP[*cur*] starting at NEXT[*LE*] and ending at *cur*. Note that we "severed" the link between *cur* and *RE* (we "extracted" the family from the list NEXT).

Sort the family

Note that sorting the family according to the letter at position *type* is the same as sorting links of an internal node of a suffix tree. We will discuss the actual sorting separately. We are assuming that *from* refers to the head of the family, while *to* to its last member. Prior to sorting the family, we must remember the LCP[*to*] value, thus *last* ← LCP[*to*]. After the sorting of the family, we must modify the LCP accordingly:

```
for i ← from to to
  if LCP[i] < type then
    LCP[i] ← type
LCP[to] → last
```

Flatten the family

As indicated, some nodes in the NEXT list might have "vertical" tails. At this stage we "flatten" the family so there are no "vertical" tails any more. The process is simple: if NEXT[a] = b, then we make NEXT[a] to be the first element in the "vertical" tail, while NEXT[c] $\leftarrow b$, where c is the last element in the "vertical" tail. Thus:

for $i \leftarrow from$ **to** to
  if TAIL[i] $\neq 0$ **then**
    $b \leftarrow$ NEXT[i]; NEXT[i] $\leftarrow$ TAIL[i]
    NEXT[TAIL[TAIL[i]]] $\leftarrow b$
    TAIL[TAIL[i]] $\leftarrow 0$; TAIL[i] $\leftarrow 0$

Verticalize the family

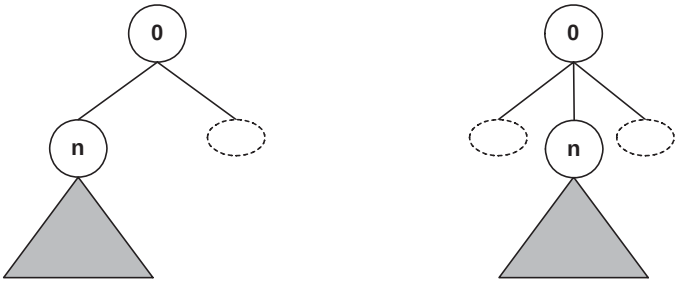
To prevent resorting or retraversing the family which just has been flattened during the subsequent sort (of the family this family is nested in), we leave only the head of the family in the NEXT list, and make the rest of the family into a "vertical" tail of the head. Thus, in all subsequent sorts only the head will be used and thus further traversal of the family is prevented.

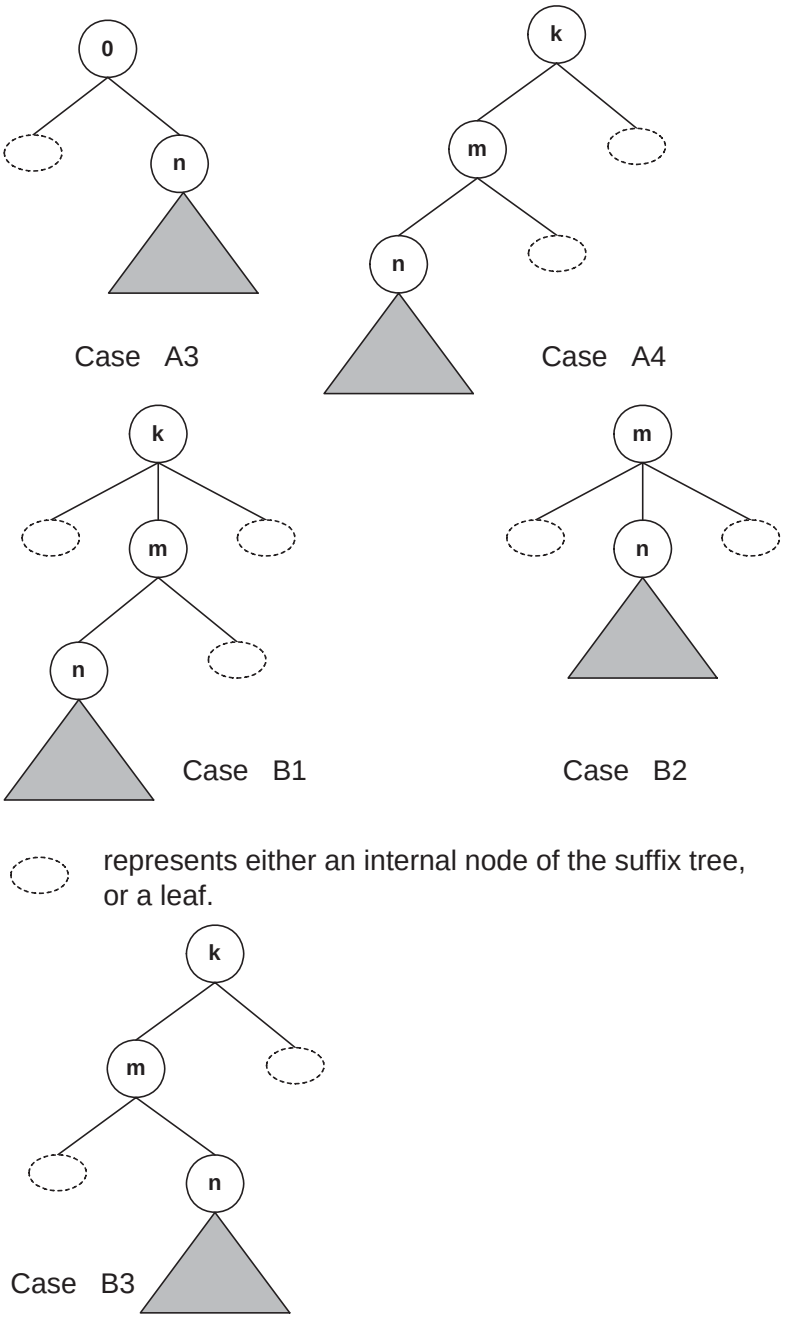
TAIL[$from$] = NEXT[$from$]
NEXT[$from$] $\leftarrow 0$
TAIL[TAIL[$from$]] $\leftarrow to$

multipop()

As a technicality, in its first invocation **multipop()** returns **true** . Thus, we can assume, that we just finished processing a family of type *type*. We have to decide if we continue with the scan, pop the stack, or process another family. The role of ***multipop***() is to make all these decisions. It returns **true** if the scan is to continue, or **false** if the scan is to terminate.

What situations can happen is best visualized on the suffix tree — the grey triangle represents the family of links that was just sorted. There are 7 possible cases that we denote A1, ..., A4, and B1, ..., B3. Cases A1, ..., A4 concern situations when only one item is on the stack (representing the family we just sorted), while cases B1, ..., B3 concern situations when more than one item are on the stack. The schematic depiction of the cases follows:





The variable $famend$ represents the "pointer" to the very last element in the family just processed.

Cases A1, ..., A3
These are treated alike and recognized alike. The recognition is based on the fact that the stack has only one item and LCP$[famend] = 0$. The action is to pop the stack, forward the scan and then the scan is continued:

$Pop()$; $prev \leftarrow cur$; $cur \leftarrow$ NEXT$[cur]$
if cur=0 **then return false**
$type \leftarrow$ LCP$[cur]$
while $type = 0$ **do**
  $prev \leftarrow cur$; $cur \leftarrow$ NEXT$[cur]$; $type \leftarrow$ LCP$[cur]$
  if type=0 **then**
    $prev \leftarrow cur$; **return false**
$Push(prev)$
return true

Case A4
The recognition is based on the fact that the stack has only one item and LCP$[famend] > 0$. The action is not to pop the stack (as the n-family just processed starts at the same position as the m-family to be processed), the scan is forwarder and then the scan is continued, but the type is decreased accordingly (to m):

$type \leftarrow$ LCP$[famend]$
$prev \leftarrow cur$
$cur \leftarrow$ NEXT$[cur]$
return true

For cases B1, ..., B3 we have to determine $type1$, the type of the family that is on the top of the stack:

if $Top_1() = 0$ **then**
  $type1 \leftarrow$ LCP$[start]$
else
  $type1 \leftarrow$ LCP[NEXT$[Top_1()]$]

Case B1
The recognition is based on the fact that the stack has more than one item and LCP$[famend] > type1$. The action is not to pop the stack (as the n-family just processed starts at the same position as the m-family to be processed), the scan is forwarded and then the scan is continued, but the type is decreased accordingly (to m):

$type \leftarrow$ LCP$[famend]$
$prev \leftarrow cur$
$cur \leftarrow$ NEXT$[cur]$
return true

Case B2
The recognition is based on the fact that the stack has more than one item and LCP$[famend] = type1$. The action is to pop the stack, decrease the type, forward the scan and then the scan is continued:

$Pop()$
$type \leftarrow type1$
$prev \leftarrow cur$
$cur \leftarrow$ NEXT$[cur]$
if $cur = 0$ **then return false**
return true

Case B3
The recognition is based on the fact that the stack has more than one item and LCP[$famend$] < $type1$. The action is to pop the stack, decrease the type, without moving forward the scan and then the scan is continued:

$Pop()$
$type \leftarrow type1$
return true

This concludes the description of the algorithm. It is rather straighforward to check that the algorithm (without the actual sorting of the families) requires $O(n)$ steps. The additional memory requirements are n words for the array NEXT[ ] and $\leq n$ words of memory for STACK. Of course, some additional memory will be required for the actual sorting of the families: if the number of distinct characters in the input string is $\leq n/2$, then we need $\leq 3n/2$ words of memory for STACK and for sorting (n for STACK and $\leq n/2$ for sorting). If the number of distinct characters in the input string is $> n/2$, then we need $\leq 3n/2$ words of memory for STACK and sorting ($< n/2$ for STACK, and $\leq n$ for sorting). Thus, **the algorithm presented requires in total** $\leq 2.5n$ **words of working memory for the process and the sorting**.

C code for Algorithms 1–3 and powerpoint illustration of Algorithms 2–3 are available at [F05].

From the presentation of the algorithm it is clear that sorting the suffix array is as complex as sorting links in the corresponding suffix tree. Thus, the following discusion applies to both suffix trees and suffix arrays. When we are to sort a family of size k (or k links of an internal node in the suffix tree), no matter what permutation is given, it can be sorted in $O(n)$ time using a bucket sort. However, this may lead to non-linear sorting time for the whole array (or the whole tree). If the alphabet is fixed, of course the sorting will be linear. But also for some "mild" permutations the sorting will be linear as well. This leads us to investigate an interesting computational property of permutations that we call the suborder complexity of the permutation:

The ***suborder complexity*** β of a permutation p of n, denoted $\beta(p)$, is defined to be the minimal β such that for any $2 \leq k \leq n$, it takes at most βk steps to order any subset of n of size k. Note that $\beta(p) \leq \log n$ as any subset of n of size k can be sorted in $\leq k \log k \leq k \log n$ steps.

It follows that

Theorem 2. *For any permutation with suborder complexity β, the suffix array of a string can be re-ordered by Algorithm 3 in $O(\beta n)$ time, where n is the length of the input string.*

Conclusions and Further Research

An interesting question that arises is what kind of permutations have small suborder complexity. Here are some examples:

- The inversion has suborder complexity 1.

- Any rotation has suborder complexity 1.
- Any permutation with β transpositions has suborder complexity β.
- Let p be a "mild" permutation, i.e. $|p(i) - i| \leq \beta$. Then p has suborder complexity 2β.
- Let p_1 on n_1 have suborder complexity β_1 and let p_2 on n_2 have suborder complexity β_2, then $p_1 \bigoplus p_2$ has suborder complexity $\max(\beta_1, \beta_2)$ (where $p = p_1 \bigoplus p_2$ is defined on n_1+n_2 by $p(i) = p_1(i)$ for $1 \leq i \leq n_1$, and $p(i) = n_1+p_2(i-n_1)$ for $n_1 < i \leq n_1+n_2$).

So the class of permutations with small suborder complexity seems quite interesting and rich enough to warrant further investigation.

Acknowledgements

The research was supported in part by the authors' research grants from the Natural Sciences and Engineering Research Council of Canada.

References

[BK03] S. Burkhardt & J. Kärkkäinen, **Fast lightweight suffix array construction and checking**, *Proc. 14th Annual Symp. Combinatorial Pattern Matching*, LNCS 2676, Springer-Verlag (2003) 55-69.

[BW94] M. Burrows & D.J. Wheeler, *A Block-Sorting Lossless Data Compression Algorithm*, Research Report 124, Digital Equipment Corporation (1994) 18 pp.

[F97] M. Farach, **Optimal suffix tree construction with large alphabets**, in *Proc. 38th Annual Symp. Foundations of Computer Science*, IEEE (1997) 137–143.

[F05] F. Franek, *C code + illustration*:

`http://www.cas.mcmaster.ca/~franek/web-publications.html`

[FS05] F. Franek & W. F. Smyth, **Sorting the suffixes of a two-pattern string**, *Internat. J. Foundations of Computer Sci.* (2005) to appear.

[FSXH03] F. Franek, W. F. Smyth, X. Xiao & J. Holub, **Computing quasi suffix arrays**, *J. Automata, Languages & Combinatorics 8–4* (2003) 593–606.

[IT99] H. Itoh & H. Tanaka, **An efficient method for in memory construction of suffix arrays**, *Proc. String Processing & Information Retrieval Symp.*, IEEE (1999) 81–88.

[KLAAP01] T. Kasai, G. Lee, H. Arimura, S. Arikawa & K. Park, **Linear-time longest-common-prefix computation in suffix arrays and its applications**, *Proc. 12th Annual Symp. Combinatorial Pattern Matching*, LNCS 2089, Springer-Verlag (2001) 181-192.

[KA03] P. Ko & S. Aluru, **Space Efficient Linear Time Construction of Suffix Arrays**, *Proc. 14th Annual Symp. Combinatorial Pattern Matching*, LNCS 2676, Springer-Verlag (2003) 200–210.

[KMR72] R. M. Karp, R. E. Miller & A. L. Rosenberg, **Rapid identification of repeated patterns in strings, trees and arrays**, *Proc. 4th Annual ACM Symp. on Theory of Computing* (1972) 125–136.

[KSPP03] D. K. Kim, J. S. Sim, H. Park, & K. Park, **Linear-time Construction of Suffix Arrays**, *Proc. 14th Annual Symp. Combinatorial Pattern Matching*, LNCS 2676, Springer-Verlag (2003) 186–199.

[KS03] J. Kärkkäinen & P. Sanders, **Simple Linear Work Suffix Array Construction**, *Proc. 30th International Colloqium on Automata, Languages and Programming*, LNCS 2719, Springer-Verlag (2003) 943–955.

[LS99] N. Jesper Larsson & K. Sadakane, *Faster Suffix Sorting*, Technical Report LU-CS-TR:00–214, Lund University (1999) 20 pp.

[MM90] U. Manber & G. Myers, **Suffix Arrays: A new method for on-line string searches**, *Proc. First ACM-SIAM Symp. on Discrete Algs.* (1990) 319–327.

[MM93] U. Manber & G. Myers, **Suffix Arrays: A new method for on-line string searches**, *SIAM J. Computing 22* (1993) 935–948.

[M05] M. Maniscalco, *MSufSort*:

`http://www.michael-maniscalco.com/`

[MF04] G. Manzini & P. Ferragina, **Engineering a lightweight suffix array construction algorithm**, *Algorithmica 40* (2004) 33–50.

[PST05] S. J. Puglisi, W. F. Smyth & A. Turpin, **The performance of linear time suffix sorting algorithms**, *Proc. Data Compression Conf. '05* (2005) to appear.

[S00] J. Seward, **On the performance of BWT sorting algorithms**, *Proc. Data Compression Conf. '00* (2000) 173–182.

[SS05] K. Schürmann & J. Stoye, **An incomplex algorithm for fast suffix array construction**, *Proc. 7th Workshop Algorithm Engineering & Experiments* (2005) to appear.

[S03] B. Smyth, *Computing Patterns in Strings*, Pearson Addison-Wesley (2003) pp. 423.

Reordering Finite Automata States for Fast String Recognition

E. Ketcha Ngassam[a], Derrick G. Kourie[b], and Bruce W. Watson[b]

[a] School of Computing, University of South Africa,
Pretoria 0003, South Africa
[b] Department of Computer Science, University of Pretoria,
Pretoria 0002, South Africa

e-mail: [a]ngassek@unisa.ac.za, [b]{dkourie, bwatson}@cs.up.ac.za

Abstract. The spatial and temporal locality of reference on which cache memory relies to minimize cache swaps, is exploited to design a new algorithm for finite automaton string recognition. It is shown that the algorithm, referred to as the state reordering algorithm, outperforms the traditional table-driven algorithm for strings that tend to repeatedly access the same set of states.

Keywords: Automata, Implementation, Performance, State Reordering, Cache Locality of Reference

1 Introduction

Traditionally, finite automata (FAs) are implemented using the table-driven (TD) algorithm extensively discussed in [1]. In this case, the processing time of the recognizer is memory load dependant in the sense that for automata of considerable size[1], the time taken to process a string not only depends on the length of the string but also on the time taken to do a lookup in the transition matrix.

In [2, 3], we reported on investigations based on hardcoded FAs which appeared to be faster than the TD algorithm, but only for automata of relatively small sizes[2]. Further investigations revealed that, although memory load and string length are major processing time factors, the kind of string being tested for acceptance is also critical. In effect, no matter the size of the string being tested for acceptance and the size of the automaton upon which the recognizer relies, a string that drives the automaton into a set of 'sink' states throughout the recognition process is always processed at optimum due to computer's cache memory [4]. In such kinds of strings, the hardcoded algorithm outperforms its TD counterpart. This is explained by the fact that the instructions that makeup the hardcoded algorithm always remain in cache—hence the fast processing speed. In this regard, cache memory plays an important role in determining the efficiency of FA-based string recognition algorithms.

[1] In this paper, we use automaton size to mean the number of states of the automaton. The two terms are used interchangeably.

[2] In fact experiments revealed that hardcoded algorithm is faster than TD for FAs of size up to about 360 states on an Intel Pentium 4. This was true for alphabet sizes of up to about 50 symbols.

Cache memory operation is based on what is sometimes referred to as the principle of *temporal and spatial locality of reference*. Since data/instructions are fetched from memory in blocks, the temporal locality of reference refers to the premise that there is a strong chance that the same data/instruction will be used in the near future. Similarly, spatial locality of reference refers to the premise that there is a strong chance that other data within a given block will be fetched in the immediate future. These two principles are of importance in the design and implementation of efficient algorithms. Moreover, the nature of the cache itself guarantees that data found in cache is processed faster than data residing in the main memory. Page swaps into cache occur when data being sought is not in cache and the cache is full. In this case, a policy such as that of LRU (least recently used) data is normally used to determine what is to be swapped out. For more information on cache, refer to [5].

In this paper, we provide an alternative algorithm for string recognition, referred to as the State Reordering (SR) algorithm that makes use of the spatial and temporal locality principles. The algorithm reorders the states of the original automaton according to the string being processed. Only states needed are reorganized in memory.

In certain circumstances, the reordering increases the probability of reusing the chunks of data already present in the cache. The evidence suggests that our algorithm will outperform the TD algorithm for large automata when processing long sequences that exercise a limited number of states. The provision is that the strings are long enough to amortize the cost of reordering the states. This would be the case, for example, in a network intrusion detection system, where a continuous stream of data is being processed by an FA-based system.

The structure of the rest of this paper is as follows. In Section 2 we present and explain the SR algorithm. Section 3 assesses it from a theoretical perspective. Section 4 deals with the experimental comparison of SR and TD. Finally, in section 5, the conclusion and further directions for this work are offered.

2 The State Reordering Algorithm

In this section, we present the new SR algorithm and provide a theoretical analysis of the algorithm based on strings of considerable length. The conditions under which the algorithm appears to be most efficient are also discussed. We further the analysis by providing a class of strings that can benefit from our algorithm provided that the set of states visited remains unchanged.

proc $tdRecognizer(table, inString)$
$; state, index := 0, 0$
do $(index < inString.length()) \wedge (state \geq 0) \rightarrow$
    $state, index := table[state][inString[index]], index + 1$
od

Figure 1: Table-driven string recognizer

It is clear from the pseudocode of the TD algorithm (see Figure 1) that its access

to the transition table in memory is entirely dependent on the string being examined. Since data is fetched by the processor from memory in chunks, the arbitrary organization of the table's entries results in frequent cache misses in the next cycle. Put differently, the probability of finding the desired datum from the cache is relatively low. The processor is then forced to perform a page swap in order to get the desired entry. This approach may result in inefficiencies when the table is considerably large.

Figure 2 provides a high-level specification of the SR algorithm. Just as in figure 1, the transition table and the input string ($inString$) are provided as parameters. Also provided as parameters are: the start address ($start$) where information about reordered states is stored; and an indication of the amount of space to be reserved for each reordered state ($size$). By a reordered state, we mean a state (as represented by a row in the original transition table) whose information have been copied (and modified — see below) into a specially reserved place in memory, indicated by the dynamic two-dimensional array, $srTable$. The main loop consists of an alternation- (i.e. if-) statement, and an assignment statement to increment the value of the current index into $inString$. The loop condition corresponds identically to that of the TD algorithm. The alternation statement has two guards: the first deals with a transition to the next state when the current state has not yet been reordered; and the second deals with a transition from a reordered state.

The algorithm uses an auxiliary array, $srMap$. The invariant of the algorithm's main loop:

$\forall i : [0, n) \cdot srMap[i] = k \wedge k \geq 0 \Leftrightarrow k \in [0, pos) \wedge isReordered(i, k, index - 1)$

articulates the nature of $srMap$, namely that the i^{th} entry of $srMap$ is a positive value, k, if and only if k indexes an entry in $srTable$ (i.e. $k \in [0, pos)$) and that "the k^{th} entry is a reordered state that corresponds to the i^{th} row in the original transition table". The predicate $isReordered(i, k, index - 1)$ is an assertion that corresponds to the words in quotes in the previous sentence, as will be discussed below.

The variable pos holds the index of the next $srTable$ entry to be created in memory. Thus, the first statement of the first guarded command assigns pos to $srMap[state]$, where $state$ is the current state, and the next symbol to be accessed is $inString[index]$. The variable $nextB$ points to the next memory address where space for the entry $srTable[pos]$ is to be allocated. The second statement of the first guarded command allocates the required memory for the $srTable[pos]$ entry, and the third statement copies the transition table values for row $state$ over into a row at $srTable[pos]$.

However, each entry $srTable[k][j]$ is required to have the following property: if its value, say m, is less than the total number of states, n, this should be construed to mean that if symbol j is encountered when in reordered state k, then a transition is to be made to state m where m is *not* a reordered state. However, if m is indeed greater or equal to the total number of states, then this should be construed to mean that the transition in reordered state k upon encountering symbol j is to the reordered state in $srTable[m - n]$. Thus, each time a reordered state is added into memory, it is necessary to check all reordered state entries to re-establish this property. An inner double loop in the first guarded command achieves this objective. Note that the predicate $isReordered(i, k, index - 1)$ is consistent with this property required of $srTable$ entries. It relies on the existence of a set $visited(p)$ which designates the set of all states visited when recognizing the first p elements of the string $inString$.

```
{Assume n is the number of states and a is the alphabet size}
proc srRecognizer(table, inString, start, size)
; srMap[0..n − 1] : = −1
; nextB, state, index, pos : = start, 0, 0, 0
{ Invariant ≜ (∀i : [0, n) · srMap[i] = k ∧ k ≥ 0 ⇔
              k ∈ [0, pos) ∧ isReordered(i, k, index − 1)
  isReordered(i, k, p) ≜ ∀j : [0, a) · m = srTable[k][j] ⇒
              ((m < n ∧ table[i][j] = m ⇔ m ∉ visited(p))
               ∨ (m ≥ n ⇔ ∃r : [0, n) · srMap[r] = m − n ∧ r ∈ visited(p)))
}
;do (index < inString.length() ∧ state ≥ 0) →
    if state < n →
       srMap[state] : = pos
       ; srTable[pos] : = malloc(nextB, size)
       ; srTable[pos][0..a − 1] : = table[state][0..a − 1]
       ; k, j : = 0, 0
       do k ≤ pos →
          do j < a →
             m : = srTable[k][j]
             ; if m < n ∧ srMap[m] < 0 → skip{m ∉ visited(index)}
             [] m < n ∧ srMap[m] ≥ 0 → srTable[k][j] : = srMap[m] + n
             [] m ≥ n → skip{m already updated}
             fi
             ; j : = j + 1
          od
          ; k : = k + 1
       od
       ; state, pos, nextB : = srTable[pos][inString[index]], pos + 1, nextB + size
    [] state ≥ n → state : = srTable[state − n][inString[index]]
    fi
    ; index : = index + 1
od
```

Figure 2: The state reordering string recognizer

The double loop is followed by assignments to update *state*, *pos* and $nextB$. If this new value of *state* turns out to be in the interval $[0, n)$ then it represents a transition to a non-reordered state, and will be dealt with in the next loop iteration by the first guard, in the way just described. However, if *state* turns out to be $\geq n$ then it will be dealt with in the next iteration by the second guard.

This second guard uses $srTable$ to perform the recognition of the string being tested for acceptance, as if it were the transition table in the conventional TD algorithm, but correcting, of course, for the offset by n in the *state*'s value. In fact, at every iteration of the loop, whenever the next state is a reordered one, then this

second guard's statement is executed, followed by the final statement to increment the *index* value.

The SR algorithm is thus subdivided into two parts: the *reordering* section, represented by the first guard's body, in which a state that has not been created is reordered—i.e. inserted into the *srTable*; and what we shall call the *hot-spot* section, represented by the second guard.

At first sight, the SR algorithm might appear to be less efficient than the TD version, due to the various tests that have to be made at each iteration as well as to the work done to create new "reordered" states. The SR algorithm would obviously be at a disadvantage in cases where, for a relatively large number of loop iterations, the *reordering* path is followed, since the time taken to allocate and copy memory will hamper the overall processing time. However, as a result of reordering, previously used states are organized contiguously in memory in the same order in which they are first traversed. This could be advantageous if it reduced the number of cache misses in iterations where the *hot-spot* is executed. We defer further discussion about these matters to section 3

A practical example of the SR algorithm is shown in the subsection below.

2.1 An illustrative example

Consider an automaton $M(s_0, \Sigma, Q, F, \delta)$ where $s_0 = 0$, $\Sigma = \{a, b, c\}$,
$Q = F = \{0, 1, 2, 3, 4, 5, 6\}$, and δ defined by a two-dimensional array, given below. This automaton is *partially* represented in Figure 3, in that it only shows transitions that will be followed when the string *abcbaabcbaabcba* is being recognized. The strings

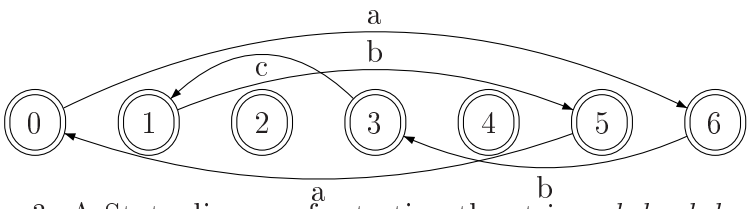


Figure 3: A State diagram for testing the string *abcbaabcbaabcba*

abcbaabcbaabcba can be processed using the SR algorithm as follows:

Initial phase:

After initialization the following holds:
$table = \{\{6, 3, 1\}, \{2, 5, 4\}, \{1, 1, 2\}, \{3, 2, 1\}, \{4, 6, 0\}, \{0, 1, 3\}, \{1, 3, 5\}\}$
(Thus, $\delta(0, a) = 6$, $\delta(0, b) = 3$, etc.)
$inString = abcbaabcbaabcba$
$inString.length() = 15$
$srMap = \{-1, -1, -1, -1, -1, -1, -1\}$.
$nextB, state, index, pos := start, 0, 0, 0$

The first iteration:

At this stage, all the conditions to enter the loop are satisfied. Therefore, the loop is executed. A test is made on $srMap[state]$ to see whether the state has been created or not. For $state = 0$, the first guard is selected and a new state has to be created in memory. This results in the following:
$srMap[0] = 0$, that is the old state 0 will occupy the first position in the new memory space.
The variable *size* represents the memory required to store a state. It depends on the alphabet size (3 for the present example).
The instructions: $srTable[pos] := malloc(nextB, size)$ and $srTable[0][0..2] := table[state][0..2]$ are then executed to produce $srTable = \{\{6, 3, 1\}\}$
The double loop tests whether any entry in the $srTable$ has been reordered to date. None has, so $srTable$ is left unaltered.
The new value of *state* is 6, *pos* becomes 1 and *index* becomes 1

Later iterations

Suppose the substring *abcba* has already been processed. If the double inner loop was not part of the algorithm then $srTable$ would simply be the following:
$\{\{6, 3, 1\}, \{1, 3, 5\}, \{3, 2, 1\}, \{2, 5, 4\}, \{0, 1, 3\}\}$. However, the double inner loop has to make sure that the entries in the new location are distinguished from those of the old location. Therefore, to avoid conflict of states, the double inner loop adds $n = 7$ to all reordered states.
This results in: $srTable = \{\{8, 9, 10\}, \{10, 9, 11\}, \{9, 2, 10\}, \{2, 11, 4\}, \{7, 10, 9\}\}$.
Therefore, for the processing of the string up to this point, only four of the six automaton states were visited. It can easily be seen that the remaining part of the string, that is *abcbaabcba*, involves the traversal of these reordered states only. Thus the remaining string is processed at *hot-spot*.

Before testing the SR algorithm empirically, it is of interest to assess theoretically how the SR and TD algorithms are likely to perform relative to one another. Such task is undertaken in the following subsection.

3 A theoretical assessment

In cross-comparing these algorithms, we rely on the fact that data in cache is processed faster than the data that is in main memory. Furthermore, when data is organized in a contiguous fashion and data items are accessed sequentially, the number of page swaps is minimized. By contrast, when data is accessed in a disorganized or random fashion, the number of cache swaps is high.

Now, as a matter of fact, ultimately neither the TD nor the SR algorithms can of themselves directly influence the way in which cache is used. They are "victims", as it were, of the strings that they are required to recognize. The following is a broad classification of the kinds of scenarios that could arise.

1. If an input string continuously drives an algorithm through a relatively small number of states such that these all remain permanently in cache, then both algorithms function optimally. Even if the input string is relatively long, the

time taken to process a single symbol is optimized. Of course, in such a case, the SR algorithm is a poor one, since it needlessly incurs the initial setup cost during the reordering phase.

2. If the input string drives an algorithm through a somewhat larger number of states, such that cache swaps have to be made, then the question is whether these cache swaps are at a minimum. Again, this behaviour is entirely dependent on the input string.

   (a) Pathological strings could be constructed to induce worst case behaviour for both the TD and SR algorithms, where as many string symbols as possible induce a transition to a state that is not currently in cache.

   (b) Likewise, well behaved string examples could be constructed where state transitions are nicely ordered to progress from row to row in the original transition table.

   In both these extreme situations, TD would perform better than SR, since SR would again incur, without any real gain, the state reordering setup cost.

3. Under the previous scenario (i.e. where a large number of states are traversed), the SR algorithm could potentially acquire an advantage over the TD algorithm if the input string exhibited the following characteristics:

   (a) the string tended to repeatedly exercise the same subset of states; where

   (b) these states were fairly widely distributed over the transition table rows, thus causing many cache misses under TD; but

   (c) where the states were contiguously placed in $srTable$ because the order of their initial usage reflected their later usage and order.

   It is easy to see that under these circumstances, the hot-spot of the SR algorithm would repeatedly be exercised in a way that minimized cache swaps, while the TD algorithm would incur a high number of cache swaps.

The claim made in 3 is rather general. It does not attempt to quantify how many reorderings should take place, how many times the hot-spot should be exercised, how long the input string should be, how rows in the transition table should be ordered, etc. Clearly all of these factors could influence the extent to which SR improves over TD. Indeed, at this point, it is not even clear whether, under practical conditions, the cost of state reordering is ever really likely to pay off. In the next section, experiments are described that offer some insights into these matters.

4 Experiments and Results

Various experiments were conducted on a 512MB Intel Pentium 4 machine, having two levels of cache memory (L1 and L2). The L1 data cache has a capacity of 8KB, with a speed of 2ccs. The L2 cache is bigger and can hold up to 256KB of data and instructions with a relative speed of approximately 6ccs. Data is fetched from memory in chunks of 64 bytes. During initial program execution, if reference is made

to a data item outside of cache, another chunk is fetched until both L1 and L2 cache are full. A subsequent fetch of data not residing in either cache, results in a page swap of memory data with data in the lowest cache. The data to be swapped out from cache is determined by the "Least Recently Used Data" policy.

The SR algorithm was implemented in the NASM assembly language under the Linux operating system. The TD algorithm was originally implemented in C++, with the optimizer (O3) turned on. Its NASM implementation was also provided after several early experiments. The intention was to ensure that the SR algorithm did not enjoy some hidden advantage because of being implemented in an assembler language. It turned out that the NASM version of TD was indeed slightly faster than its C++ implementation for automata larger than about 3000 states. However, the difference was so small that the overall results of our findings apply, no matter which TD implementation is considered.

For the present experiment, 100 automata of size $n = 125, 250, 325, \cdots, 12500$ were generated, based on 10 alphabet symbols. The transition table of each automaton was randomly constructed in the following sense:

- Firstly, for each row, $i : [0, n-2]$, a column $j : [0, a)$ (corresponding to some alphabet symbol) is randomly selected. This column is assigned the next state transition value $i + 1$. This ensures that there is at least one string of length $n - 1$ that will traverse every state of the FA. We shall refer to this string as the *root* string of the particular automaton.

- Next, all remaining cells of the table are assigned a random value in the range $[0, n-1]$.

Considered graphically, this means that each node in the FA graph has a transition to the next state on some random symbol, as well as a transition on each of the remaining 9 alphabet symbols to some random state.

For each automaton, a random string of size $n - 1$ was generated. This was replicated 4 times to produce an input string of length $4n - 4$ consisting of 4 identical segments. Each of the algorithms was required to use the randomly generated automaton of size n to recognize such a string, resulting in 100 runs of each algorithm.

Before discussing the timing results, consider the information presented in table 1. The table gives an overview of the rate at which state reordering was found to occur when the SR algorithm was run. Data is given as a percentage of the total number of states in each particular run. The first column relates to the full string that was processed, the second column, to the first segment, etc. Thus, after processing the first segment, on average a little more than 60% of the states are reordered. Note that these observations lie in a fairly narrow band, between about 57% and 63%. As a matter of fact, when the number of reordered states is plotted against the automaton size (not provided here), a very distinct linear trend is observed. However, in the case of segments 2 to 4, no obvious trend is observed in relation to automaton size. Nevertheless, the average number of reordered states declines steadily from about 16% in the case of segment 2, to almost 0% in the case of segment 4. Overall, about 80% of states were reordered, on average.

These results are broadly in line with expectation. In processing the first $n - 1$ symbols, roughly 60% of the states are reordered, meaning that they are located

	Full String	Segment 1	Segment 2	Segment 3	Segment 4
Maximum	95.20%	62.67%	24.80%	9.17%	2.78%
Minimum	62.42%	56.80%	0.40%	0.00%	0.00%
Average	79.06%	61.29%	16.11%	1.60%	0.06%

Table 1: Rate of Reordered State Generation

contiguously in memory in order of first usage. In this sense, the data is optimized in terms of the spatial locality of reference principle. Later segments trigger progressively fewer state reorderings, and consequently spend more time in the hot-spot part of the code. If these later segments were to traverse the reordered states in *exactly* the same sequence as the first segment, then the probability would be relatively high of accessing spatially localized data, and hence of triggering few cache swaps. Of course, this will only happen in the unlikely event that segment 2 (and therefore also 3 and 4) happen to start off in state 0.

The experiment above has not been designed to specifically generate this "best case" scenario. Rather, it is far more likely that these later segments will start off in some other random state. Nevertheless, on the evidence of table 1, an increasingly large proportion of segment 2 to 4 processing is via the hot-spot. In fact, even in the case of the first segment's processing, about 40% of the iterations were through the hot-spot. Whether this translates into time gains as a result of frequently accessing spatially localized data (and therefore having fewer cache swaps), cannot be predicted *a priori*. To this end, we require the timing data derived from running the respective algorithms.

For the purposes of recording timing data, each algorithm was invoked 50 times for each set of input, and the processing time was recorded in clock cycles (ccs). For further analysis, we relied on the minimum of these 50 observations. (This was because the experience of earlier studies, which had shown that occasionally, outlier data is generated that distorts the average and that is apparently attributable to OS and CPU overheads.)

The results showed that state reordering is too expensive to provide such a short-term payoff. In fact, the cost is at least 100 times than that of making a transition. In order to gain any advantage from the spatial and temporal locality of reference of the reordered states, and thus amortize the cost of reordering, the hot-spot would have to be exercised much more frequently than was done by the strings of length $4n - 4$.

A further experiment was therefore conducted to probe the best case scenario—one in which reordered states are traversed in the same order as they were generated. This was done by essentially repeating the previous experiment with the following modifications: strings were now of length $2n - 2$ instead of $4n - 4$; it was ensured that when the n^{th} symbol was encountered then the FA would be in state 0; and only the time taken to process the second group of $n - 1$ symbols was measured. Gnuplot was used to plot the graphs of number of states against time for both TD and SR algorithms. The results are provided in figure 4.

The graph shows that the TD processing time is super-linear in the size of the automaton. Although the SR trend appears to be close to linear, there is, in fact, a slight suggestion of superlinearity here as well. It is clear that in this case, SR enjoys

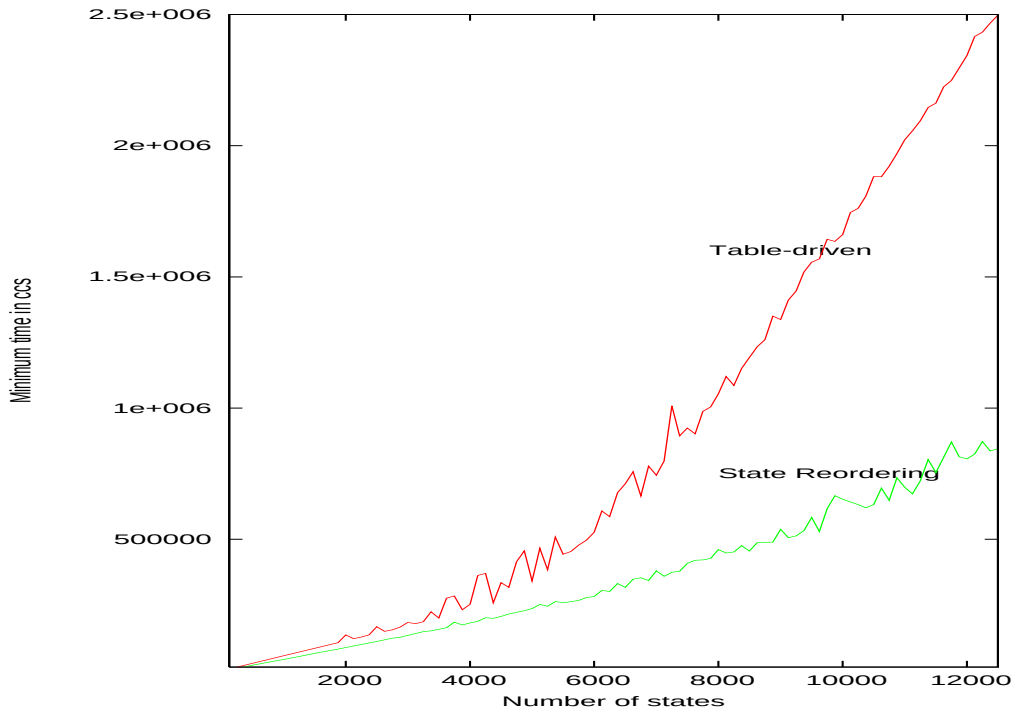


Figure 4: SR and TD Performance: Input String is Two Repeated Segments of Length $n-1$. Time is for Second Segment Only

a definite time advantage over TD, due to optimal cache utilization. For example, at about 9000 state transitions, the SR algorithm is about 60% faster than the TD algorithm. The graph may be thought of as SR's best case asymptotic behaviour. On this evidence, therefore, state reordering is a feasible strategy under conditions that approximate those discussed in section 3, item 3. Given that more recent hardware platforms have been placing increasing emphasis on additional cache memory[3], the gains obtained by optimally exercising cache are likely to increase.

5 Conclusion and Future Work:

In this paper, we have discussed the design of an algorithm for FA string recognition that attempts to leverage an advantage from the fact that cache memory relies on the principle of spatial and temporal locality of reference. Our experiments have suggested that the SR algorithm could gain an advantage over the traditional TD algorithm for long string sequences that tend to revisit hot-spot states in a certain order.

Two application areas that immediately suggests themselves as potentially worth exploring are DNA analysis and network intrusion detection. In the first case, one of the contemporary challenges is the identification of so-called micro- and/or mini-satellites (generically called approximate tandem repeats) in DNA strings. Here, what is sought is repeated approximate patterns in the string. The notion of "repetition"

[3]For example, the L2 cache of Intel's Prescott-2M Pentium 4 chip, released in February 2005, has 2048kB, while the Intel Intanium 2 processor, targeted for release in November 2005, will have a 3MB of L3 cache.

intuitively corresponds to the idea of processing within a hot-spot, as discussed earlier. In the latter case, one would imagine that scanning a stream of network data for security breaches involves, for the most part, the traversal of hot-spot states that should quickly pass the data down the line. Again, this seems like a possible application domain for the SR algorithm. However, a fuller investigation of appropriate application domains for SR is a matter left for future research.

The algorithm was implemented in NASM on an Intel Pentium 4 machine. Intel offers many data prefetching instructions for performance enhancement [6] that have not been used in the algorithm. These instructions should be analyzed in the future in the hope of speeding up even further the SR implementation.

The algorithm suggested in this paper is part of a set of algorithms under investigation. The aim is to package these in a dynamic framework for implementing FAs with a view to enhancing performance [4]. We are currently investigating a mixed-mode implementation as an alternative to the SR and hardcoded implementations explored to date. Once all the algorithms under investigation have been tested, the final design of the dynamic framework will be proposed as well as a toolkit for efficiently processing FAs.

References

[1] Ketcha Ngassam, E. *Hardcoding Finite Automata.* MSC Dissertation. University of Pretoria, 2003.

[2] Ketcha Ngassam, E., Watson, B. W. and Kourie, D.G. *Preliminary Experiments in Hardcoding Finite Automata.* Poster paper, CIAA, Santa Barbara, 299–300, September 2003.

[3] Ketcha Ngassam, E., Watson, B. W. and Kourie, D.G. *Hardcoding Finite State Automata Processing.* SAICSIT, Johannesburg, 111–121, September 2003.

[4] Ketcha Ngassam, E., Watson, B. W. and Kourie, D.G. *A Framework for the Dynamic Implemenation of Finite Automata for Performance Enhancement.* Prague Stringology Conference, Prague, August 2004.

[5] Hannessy, J. L., Patterson D. A. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, 3rd Edition, 2003.

[6] Intel Corporation. *The Intel Opimization Reference Manual.* http://www.intel.com/design/pentiumiii/manuals/, 2002.

[7] Thompson, K. *Regular Expression Search Algorithm.* Communications of the ACM. Volume 11, No 6, 323–350, 1968.

[8] Knuth, D.E., Morris Jr., J.H. and Pratt, V. R *Fast Pattern Matching in Strings.* SIAM J. Comput. Volume 6, No 1, 323–350, 1977.

[9] Yao, A C. *The Complexity of Pattern Matching for a Random String.* SIAM J. COmput., 8(3),pp. 368-387, 1979.

[10] Gerber, R., *The Software Optimization Cookbook: High-Performance Recipes for the Intel Architecture*. Intel Corporation, 2002.

Backward Pattern Matching Automaton

Jan Antoš and Bořivoj Melichar

Department of Computer Science & Engineering
Faculty of Electrical Engineering
Czech Technical University
Karlovo nám. 13, 121 35 Prague 2

e-mail: {antosj,melichar}@fel.cvut.cz

Abstract. We present a new algorithm to solve a large number of backward pattern matching problems. This algorithm is specified by the theory of finite automata. The algorithm is based on the utilization of a formal tool called "Backward Pattern Matching Automaton", which we specify in this paper. Introduction of such a tool presents a formal base to the world of backward pattern matching.

Keywords: backward pattern matching, string matching, sequence matching, approximate pattern matching, subpattern matching, don't care symbols, finite automata, formal tool

1 Introduction

Pattern matching (string and sequence matching) is an essential part of many applications. This discipline has been intensively studied since the beginning of the seventies and many pattern matching problems have been discovered and extensively studied. A number of new algorithms was presented. Yet these algorithms lack a common theory and are often hard to understand, evaluate and proof. One reason for such a diversity is the nonexistence of a uniform formalism needed for the specification of the problems themselves.

In 1996 a formalism was found, which allows principles of matching algorithms to be formally specified. This formalism is based on a finding, that all one-dimensional matching algorithms are sequential problems and thus can be solved by the use of finite automata [MH97a].

At the same time a classification of matching problems was presented. This classification is not (and cannot be) complete, but it classifies 192 different pattern matching problems in a six-dimensional space [MH97][1]. Together with the new formalism it resulted in an interesting fact: Having a finite automaton to describe the pattern matching problem of one string in a text, all the other 191 problems can be solved by simple operations applied to this one automaton [MH97]. Only a forward matching technique was explored in [MH97] leaving the question open, if similar operations

[1] The number of problems was further increased to 336 in the following years by the addition of approximate matching over well-ordered alphabets.

can be defined to solve all the above mentioned pattern matching problems using the backward pattern matching algorithm.

The motivation of this paper is to present a formal specification of a backward pattern matching automaton which will be used as a model in a general backward pattern matching algorithm. The algorithm itself is simple and general and is the same for any backward pattern matching problem. The only part that is changed is the model of the problem which is fed to the algorithm input. This paper specifies the algorithm and the model and shows the construction of the model for a selected problem. It is important to mention that models for other 336 problems (as well as any future ones) can be obtained from the already defined models by simple operations over the finite automata.

2 Basic Definitions

This paper uses common notions from graph and finite automata theory. Only notions not commonly used, or notions that are specific to this paper are mentioned in this section.

Definition 2.1 (Complement of symbol). *Given an alphabet A and a symbol $a \in A$, the* complement *of a according to A is the set of symbols $\overline{a} = \{s : \ s \in A, s \neq a\}$.*

Definition 2.2 (Proper prefix). *w is a* proper prefix *of P when $w \in pref(P) \ \wedge (wv \in P, v \in A^+)$ which can also be expressed as $w \in pref(P \setminus \{w\})$.*

Definition 2.3 (Move of FA). *A* move *of a finite automaton is such a change of configuration of the finite automaton, that exactly one symbol has been read from the automaton input.*

Remark. *Note the difference between a* move *and a* transition. *While a move is a change of configuration resulting from reading a symbol a transition is a relation $\vdash_M \subset (Q \times A^*) \times (Q \times A^*)$ defined as $(q, aw) \vdash_M (p, w)$ where $p \in \delta(q, a)$, $a \in A \cup \{\varepsilon\}$, $w \in A^*$, $p, q \in Q$. Because an automaton can contain ε-transitions, one move can look for example like: $(q_1, aw) \vdash (q_2, aw) \vdash (q_3, w)$ given $a \in A$, $w \in A^*$, $q_2 \in \delta(q_1, \varepsilon)$, $q_3 \in \delta(q_2, a)$.*

Definition 2.4 (Collection). *A* Ccollection *is a set, that can contain duplicates. We will use symbols [and] to mark the collection.*

Definition 2.5 (Reversed string). *Let us have string $u \in A^*$, $u = a_1 a_2 \ldots a_n$, $a_i \in A$. Then string $v = u^R$, where $v \in A^*$ and $v = a_n a_{n-1} \ldots a_1$, $a_i \in A$ is called* reversed string. *All reversed strings from a set of strings $W \subset A^*$ will be denoted by W^R.*

Remark. *A particular* substring *of a string s, where the substring starts at position i of the string s and ends at positions j (inclusive), will be denoted as $s_i \ldots s_j$.*

3 Problem Specification

3.1 Brief Introduction to Backward Pattern Matching

Backward pattern matching can greatly speed up the pattern matching process because it is capable of skipping parts of the text. Thus we can achieve time complexity

lower then $O(n)$. The main point of backward pattern matching is that the pattern is compared from the right to left. Several techniques exist, this paper is going to explore the BDM method [CR94]. The prefix of the pattern is searched for in the text. When the longest prefix is found, the position in the text is shifted accordingly. The algorithm is therefore skipping parts of the text, where no match can occur. This principle is visualized in Figure 1.

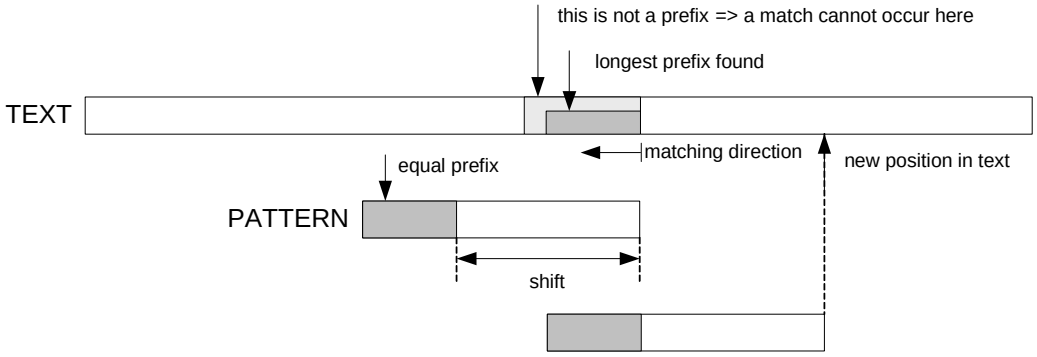


Figure 1: The backward pattern matching principle followed in this paper

3.2 Classification of Pattern Matching Problems

The classification of pattern matching problems has been described in [MH97]. This subsection presents a brief extract of the main ideas. See [MH97] for full details.

Pattern matching problems for a finite size alphabet can be classified according to several criteria. We will use six criteria for classification leading to a six-dimensional space in which one point corresponds to a particular pattern matching problem. Let us make a list of all dimensions including possible values in each dimension:

1. Nature of the pattern: string, sequence.
2. Integrity of the pattern: full pattern, subpattern.
3. Number of patterns: one, finite number, infinite number.
4. The way of matching: exact, approximate matching with Hamming distance (R-matching), approximate matching with Levenshtein distance (DIR-matching), approximate matching with generalized Levenshtein distance (DIRT-matching), approximate with Δ-matching, approximate with Γ-matching, approximate with $max(\Delta, \Gamma)$-matching.
5. Importance of symbols in pattern: take care of all symbols, don't care of some symbols.
6. Number of instances of pattern: one, finite sequence.

If we count the number of possible pattern matching problems, we obtain

$$N = 2 \cdot 2 \cdot 3 \cdot 7 \cdot 2 \cdot 2 = 336.$$

In order to make references to a particular pattern matching problem easy, we will use abbreviations for all the problems. These abbreviations are summarized in Table 1. Using this method, we can, for example, refer to exact string matching of one string as the SFOECO problem.

Instead of a single pattern matching problem we will use the notion of a family of pattern matching problems. In this case we will use symbol ? instead of a particular letter. For example SFO??? is the family of all the problems concerning one full string matching.

We will denote a pattern matching problem by symbol Θ. A pattern matching problem can be then written, for example, as $\Theta = SFOECO$ or $\Theta = SFO???$.

Dimension	1	2	3	4	5	6
	S Q	F S	O F I	E R D T Δ Γ M	C D	O C

Table 1: Abbreviations of dimension values

Remark. *The input to the pattern matching algorithm is text T and pattern set P. Because it is often difficult to define the set P, we will sometimes specify the input to the algorithm using the base pattern set P_Θ and the pattern matching problem Θ. Let us show how these relate to each other on a few examples:*

$\Theta = SFOECO, P_\Theta = \{banana\} \quad \Rightarrow \quad P = \{banana\},$
$\Theta = SSOECO, P_\Theta = \{banana\} \quad \Rightarrow \quad P = \{w : w \in fact(banana)\}.$

To further simplify the notation and to make the text more readable we will use an abbreviation of the above statements. For example:

$P = \{banana\}_{SFORCO} \quad \Rightarrow \quad P = \{w : w \in A^*, D_H(w, banana) \leq k\},$

where D_H is the Hamming distance and k is the maximum distance still considered to be a match.

4 Range of Problems Solved by This Paper

In this paper we will present a general algorithm which is capable of solving all the above mentioned problems. This algorithm should also solve any future problems. The algorithm uses the *Backward Pattern Matching Automaton (BPMA)* which is used as a model of a particular pattern matching problem. Some of these BPMA are presented here as examples. In the case of new pattern matching problems defined in the future, the only task is to define the appropriate BPMA. A very important part of this paper is to show, that these BPMA can be derived from the simplest BPMA for the SFOECO problem (exact pattern matching of one string) only by the simple operations performed over the finite automaton.

5 The Solution

The motivation is to design a simple algorithm which can be applied to a vast range of problems. Such an algorithm has to be independent of the actual problem we are trying to solve. We thus separate the pattern matching into two phases.

Phase One is the "construction phase". The input to Phase One is the type of problem specified by Θ and the set of patterns P that we want to match. The output of Phase One is the model M of the problem Θ applied to the base set of patterns P_Θ. Model M has the form of an attributed nondeterministic finite automaton. Construction of this model is different for different problems, but it has a common base: the basic pattern matching model is constructed first and then automaton operations are applied to it and the final model is derived.

Phase Two is the "matching phase". The input to Phase Two is the model M (the model of the pattern matching problem constructed in Phase one) and the text T. The output of Phase Two is the set of occurrences of patterns $p \in P$ in text T. The automaton M is repeatedly used in the matching phase and attributes of its states and transitions are evaluated. Phase Two is thus completely independent of the problem Θ.

6 Backward Pattern Matching Automaton

Each pattern matching problem can be described using its model in the form of an attributed nondeterministic finite automaton (Definition 6.1). This model is then used in the pattern matching phase.

Definition 6.1 (Attributed Nondeterministic Finite Automaton). Attributed Nondeterministic Finite Automaton (ANFA) M *is five-tuple* $M = (NFA, R, \gamma_q, \gamma_\delta, G)$ *where*

NFA	*is nondeterministic finite automaton* $NFA = (Q, A, \delta, q_0, F)$,
R	*is a finite set of attributes. Every attribute has a domain* $H(r)$ *specifying possible values of attribute* r. $R = R_q \cup R_\delta$,
γ_q	*is a mapping* $Q \times R_q \to H(r) \cup \emptyset$ *where* $r \in R_q$,
γ_δ	*is a mapping* $Q \times Q \times A \times R_\delta \to H(r) \cup \emptyset$ *where* $r \in R_\delta$,
G	*is a finite set of semantic rules of the following form:* $p.r \leftarrow g(q.r_1, \ldots, q.r_n, p.r_1, \ldots, p.r_m, t_{q,p,a}.r_1, \ldots, t_{q,p,a}.r_k)$ *where* $q.r$ *denotes a* $\gamma_q(q, r)$ *and reads as "value of attribute* r *of state* q", $t_{q,p,a}.r$ *denotes* $\gamma_\delta(q, p, a, r)$ *and reads as "value of attribute* r *of transition* $\delta(q, a) \ni p$ ", *and where* $m, n, k \in \mathbb{N}$. *At places, where no confusion arises, we will use* $t.r$ *instead of* $t_{q,p,a}.r$.

In this paper we are going to define ANFA common to the S????O pattern matching problems. We are going to call this automaton BPMA (Backward Pattern Matching Automaton). If more pattern matching problems are to be solved, there might be a need to extend its set of attributes R and/or its set of semantic rules G.

Definition 6.2 (Backward Pattern Matching Automaton). *A* Backward Pattern Matching Automaton (BPMA) M *is an attributed nondeterministic finite automaton* $M = (NFA, R, \gamma_q, \gamma_\delta, G)$ *where*

$NFA = (Q, A, \delta, q_0, F)$,
$L(NFA) = pref(P^R)$, P *is set of patterns,*
$R = R_q \cup R_\delta$,
$R_q = \{tc, plc, pf\}$,
$H(tc) = \mathbb{N}$,
$H(plc) = \mathbb{N}$,
$H(pf) = \{\textsc{True}, \textsc{False}\}$,
$R_\delta = \{ptf\}$,
$H(ptf) = \{\textsc{True}, \textsc{False}\}$,
$G = \{p.tc \leftarrow q.tc + 1$,
$p.pf \leftarrow$ ***if*** $t.ptf = \textsc{True}$ ***then*** $\textsc{True}$ ***else*** $q.pf$,
$p.plc \leftarrow$ ***if*** $q \in F \ \wedge \ p.pf = \textsc{True}$ ***then*** $p.tc$ ***else*** $q.plc$,
$t.ptf$ *is precomputed for all transitions,*
$q_0.tc \leftarrow 0$,
$q_0.pf \leftarrow \textsc{False}$,
$q_0.plc \leftarrow 0$ $\}$ *for* $q, p, t : \ \delta(q, a) \ni p, \ t \sim t_{q,p,a}$.

Let's have a BPMA and

$$(q_0, w^R z^R) \vdash^* (q, z^R), \ q \in Q, \ w, z \in A^*,$$

then we can explain the meaning of BPMA state attributes as follows:

Attribute *tc* is the acronym for *Transition Counter*. This attribute stores the number of automaton moves. This number equals the number of symbols read from the automaton input to reach the current configuration:

$$q.tc = |w|.$$

Attribute *plc* is the acronym for *Prefix Length Counter*. This attribute stores the length of some proper prefix found from the last shift operation. Because the automaton is nondeterministic and several options for $(q_0, w^R z^R) \vdash (q, z^R)$ are possible, the value of $q.plc$ does not have to be the actual longest proper prefix of w, so in the final count, we have to evaluate all of the $q.plc, q \in F$ to find the plc_{max}. The fact that $plc_{max} = |pref(P \backslash \{w\})|_{max}$ has to be assured by the way the model is built. Then we can state that:

$$q.plc \in \{|v| : v, u \in A^*, \ vu = w, \ v \in pref(P \backslash \{v\})\},$$

$$plc_{max} = max\{q.plc : q \in F\}.$$

Attribute *pf* is the acronym for *Prefix Flag*. Attribute *pf* of a state q has value $\textsc{True}$ if from a current automaton configuration every future final configuration reached indicates that a proper prefix of some pattern $p \in P$ has been found. If the value of any final state is $\textsc{False}$ it indicates, that an occurrence of a pattern has been found:

$$q.pf = \textsc{True} \quad \Rightarrow \quad \forall u \in A^*, \ \forall q_f \in F, \ \delta(q, u^R) \ni q_f : uw \in pref(P \backslash \{uw\}),$$

$$q.pf = \textsc{False} \ \wedge \ q \in F \quad \Rightarrow \quad w \in P.$$

The meaning of the BPMA transition attributes can be explained as follows:

Attribute *ptf* is the acronym for *Prefix Transition Flag.* Attribute *ptf* of transition $\delta(q, a) = q',\ q, q' \in Q,\ a \in A^*$ has value TRUE if by an associated move the automaton will move to such a configuration, that any final state reached from there will mean, that we have found a proper prefix of some pattern $p \in P$:

$$t_{q,q',a}.ptf = \text{TRUE} \quad \Rightarrow$$
$$\forall u, v \in A^*,\ \forall q_f \in F,\ \delta(q_0, v^R) \ni q,\ \delta(q', u^R) \ni q_f : uav \in pref(P \backslash \{uav\}).$$

Note, that while some string $w \in P$ can also be a proper prefix $w \in pref(P \backslash \{w\})$, the automaton mentioned above is inherently nondeterministic: both of the following situations can happen at the same time:

$$q_f \in \delta(q_0, w^R) \ \wedge\ q_f.pf = \text{TRUE}$$
$$q_f \in \delta(q_0, w^R) \ \wedge\ q_f.pf = \text{FALSE}.$$

This behavior is wanted in this case because we want to detect both situations simultaneously. We need to know that a pattern occurrence has been found and also we need to know that an occurrence of a proper prefix has been found, so we can compute the appropriate shift function.

See the following sections for examples of backward pattern matching automata.

7 The Algorithm

7.1 Definition of the Algorithm

Phase Two has as input model M of pattern matching problem Θ and the text T in which we want to perform the actual pattern matching. Phase Two performs the matching itself. It consists of the specific backward pattern matching algorithm. This algorithm is simple and unified – the algorithm is the same for all the pattern matching problems defined in 3.2 and possibly for future ones.

The backward pattern matching algorithm is described in Algorithm 1. This algorithm uses a nondeterministic pattern matching model M and therefore it has to simulate its deterministic behavior. Future work is to construct a deterministic pattern matching model and to simplify the backward pattern matching algorithm.

Also notice, that instead of a set of states, the algorithm uses a collection of states. This is required to allow the processing of one state with different attributes – this situation can happen when the automaton has two transitions for the same symbol going from state q to state p and for one transition $t_{q,p,a}.ptf = \text{TRUE}$ and for the second $t_{q,p,a}.ptf = \text{FALSE}$.

Algorithm 1:	AUTOMATON-BASED BACKWARD PATTERN MATCHING ALGORITHM
Input:	Model M in the form of Backward Pattern Matching, Automaton $M = (NFA, R, \gamma_q, \gamma_\delta, G)$, text T.
Output:	Set of numbers, each number represents a position in text T where pattern $p \in P$ occurs.
Method:	

1 $\quad position \leftarrow |P|_{min}$

```
2     offset ← 0
3     plc_max ← 0
4     Q' ← [q_0] (see Definition 2.4)
5     while position ≤ |T| do
6         Q'' ← [q: q ∈ δ(q', T_position−offset), q' ∈ Q']
7         if Q'' ≠ ∅ then
8             for ∀q ∈ Q'' do
9                 if q ∈ F ∧ q.pf = False then
10                    output(position − offset)
11                end if
12                if q ∈ F ∧ q.pf = True then
13                    plc_max ← max{plc_max, q.plc}
14                end if
15            end for
16            Q' ← Q''
17            increment offset
18        else
19            shift ← max{1, |P|_min − plc_max}
20            position ← position + shift
21            offset ← 0
22            plc_max ← 0
23            Q' ← [q_0]
24        end if
25    end while
```

The main idea of the algorithm is as follows:

1. The algorithm computes the initial position.

2. The algorithm utilizes the BPMA automaton in order to decide, if there is some pattern ending at this position, i.e. if

$$\exists x \in \mathbb{N} : T_x \dots T_{position} \ \in \ P.$$

   This event occurs if

$$\exists q_f \in F, \ \exists w \in A^* : \delta(q_0, w^R) \ni q_f \ \wedge \ q_f.pf = \text{False}.$$

   In this case, the value of x is output.

3. Simultaneously with Step 2, the algorithm also has to decide what the longest proper prefix ending at this position is, i.e. it computes $|w|_{max}$ where

$$w \in pref(P \backslash \{w\}) \ \wedge \ w = T_{position-|w|} \dots T_{position}.$$

   This $|w|_{max}$ (named plc_{max} in the algorithm) equals the following expression in BMPA:

$$|w|_{max} = max\{q.plc : q \in F\}.$$

4. When the length plc_{max} of the longest proper prefix is known, the algorithm can attempt to compute the longest safe shift. A safe shift means how much it can advance the position in the text in order not to skip any occurrence of any pattern. The trivial safe shift is 1. It is easy to see, that the longest safe shift can never be longer than the shortest pattern $p \in P$ which is $|P|_{min}$. Since we know, that there is a potential of a pattern occurring at position $position - plc_{max}$, and we know $|P|_{min}$, the shift of $|P|_{min} - plc_{max}$ will be safe. So, summarized, shift can be

$$shift = max\{1,\ |P|_{min} - plc_{max}\}.$$

   Note at this point of time, why we are using the proper prefixes in contrary to traditional prefixes. If we have found string w and $w \in pref(P)$ but $w \notin pref(P \backslash \{w\})$ then $w \in P$. The value of shift would always be 1, which is inefficient in most cases, since there is no possibility of finding another pattern starting at the position $position - |w|$ or $position - |w| + 1$.

   The longest safe shift can be longer than the one mentioned in previous paragraphs. The idea of a longer safe shift is to select the shortest pattern that can start with the prefix (or prefixes) ending at the current position (w in step 3). In most cases this number can be higher than $|P|_{min}$. Let us compute P' based on that finding:

$$P' = \{p;\ p \in P, w \in pref(p)\}.$$

   The longest safe shift is then

$$shift = max\{1,\ min\{|P|_{min},\ |P'|_{min} - plc_{max}\}\}.$$

   This optimization is not employed in the current algorithm. It should be included in future works.

5. The algorithm advances its position by the $shift$ value:

$$position \leftarrow position + shift$$

   and the algorithm repeats steps 2 through 5 of this explanation until the end of the text is reached.

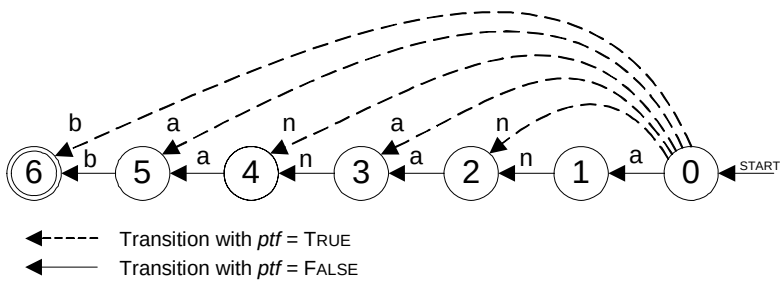


Figure 2: Transition diagram of BPMA which is a model of pattern matching problem $\Theta = SFOECO$ and pattern set $P_\Theta = \{banana\}$

7.2 Example

Let us demonstrate Algorithm 1 on a simple example. A more advanced example is given in Section 8.

Let us have a pattern matching problem $\Theta = SFOECO$, pattern set $P = \{banana\}$ and text $T = banabbababnananbananaba$. The model M of this problem is the nondeterministic pattern matching automaton given by the transition diagram specified in Figure 2. The algorithm steps are shown in Figure 3.

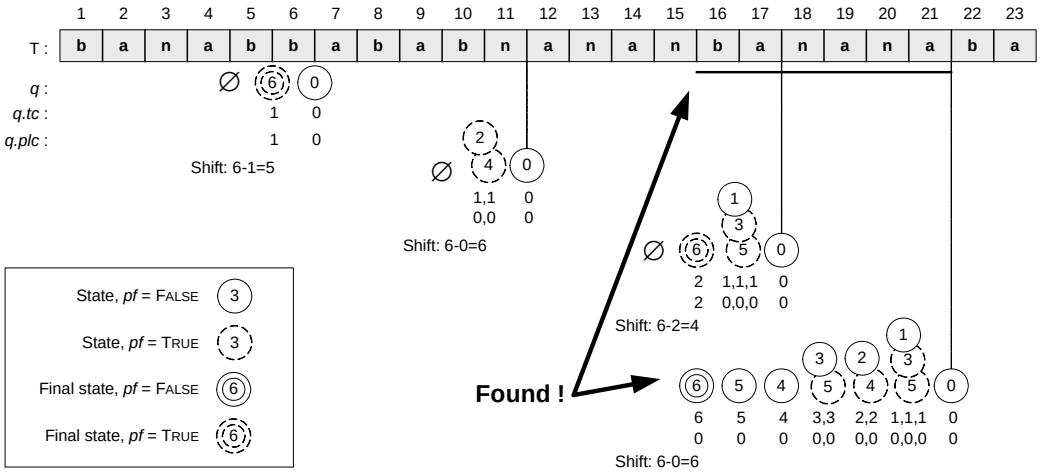


Figure 3: Steps taken during the pattern matching of $P_{SFOECO} = \{banana\}$ in text $T = banababababnananbananaba$

8 BPMA Construction

The construction Phase (i.e. Phase One) is dependent on the Backward Pattern Matching Problem solved. The output of Phase One is the uniform model of the problem. This model is in the form of a BPMA. Phase Two then uses this model to perform the actual pattern matching.

We will demonstrate the construction of such a BPMA on a selected example: Let us have a problem $\Theta = SFORCO$ (i.e. approximate R-matching of one pattern). R-matching means approximate matching where the operation "replace" is allowed. This kind of approximate matching was first explored by Hamming in [HAM50].

Let us have a base pattern set $P_\Theta = \{banana\}$ and $k = 1$. We can express P as

$$P = \{w : D_H(p, w) \leq k,\ p \in P_\Theta, w \in A^*\},$$

where k denotes the maximum distance between two patterns that we consider being equal (and thus representing a match in the text).

We first construct the *base nondeterministic finite automaton* which accepts the language $L = P^R$. We can build this automaton incrementally using the given 6D classification as an advantage: we can start with the base SFOECO problem first and then add the complexity dimension by dimension. In our case there will be only one more step necessary and it is to change the choice of value in the 4^{th} dimension: SFOECO $\rightarrow$ SFORCO.

We first build the base automaton M_1 for SFOECO problem: $P_{SFOECO} = \{banana\}$. The language accepted by this automaton is $L(M_1) = \{ananab\}$.We will use Algorithm 2. The result of this algorithm is given in Figure 4.

Algorithm 2: Construction of SFOECO base NFA
Input: Pattern p, $|p| = m$, $p \in A^*$.
Output: Deterministic finite automaton M.
Method:

1 $Q \leftarrow \{q_0, q_1, \ldots, q_m\}$
2 $\delta(q_i, p_{m-i}) \leftarrow \{q_{i+1}\}$ for all $i = 0, 1, \ldots, m-1$
3 $F \leftarrow \{q_m\}$
4 $M \leftarrow (Q, A, \delta, q_0, F)$

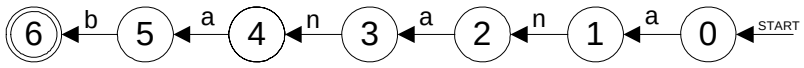


Figure 4: Transition diagram of automaton M_1, which is base NFA for $\Theta = SFOECO$ and $P_\Theta = \{banana\}$

The next step is to construct the base automaton M_2 for our chosen SFORCO problem. We use the already built automaton M_1 and modify it to recognize the language $L(M_2) = \{w : D_H(ananab, w) \leq 1,\ w \in A^*\}$. This is done employing Algorithm 3. The result is shown in Figure 5.

Algorithm 3: Construction of SF?R?O base NFA from SF?E?O base NFA
Input: Nondeterministic finite automaton $M_{SF?E?O} = (Q, A, \delta, q_0, F)$.
Output: Nondeterministic finite automaton $M_{SF?R?O}$.
Method:

1 $Q' \leftarrow \emptyset,\ F' \leftarrow \emptyset$
2 **for** $\forall l \in \langle 0, k\rangle$ **do**
3 $\quad Q' \leftarrow Q' \cup \{q_{l,i} : q_i \in Q\}$
4 $\quad \delta'(q_{l,i}, a) \leftarrow \delta'(q_{l,i}, a) \cup \{q_{l,j} : q_j \in \delta(q_i, a)\}$ for all $a \in A,\ q_i \in Q$
5 $\quad F' \leftarrow F' \cup \{q_{l,i} : q_i \in F\}$
6 **end for**
7
8 **for** $\forall l \in \langle 0, k-1\rangle$ **do**
9 $\quad \delta'(q_{l,i}, \overline{a}) \leftarrow \delta'(q_{l,i}, \overline{a}) \cup \{q_{l+1,j} : q_j \in \delta(q_i, a)\}$ for all $a \in A,\ q_i \in Q$
10 **end for**
11
12 $M_{SF?R?O} \leftarrow (Q', A, \delta', q_{0,0}, F')$

After this step we are ready to build the BPMA itself. We can use a general Algorithm 4. This algorithm constructs the BPMA from the given base NFA, where the base NFA can represent any of the problems solvable by the BPMA itself. The resulting automaton M is given in Figure 6.

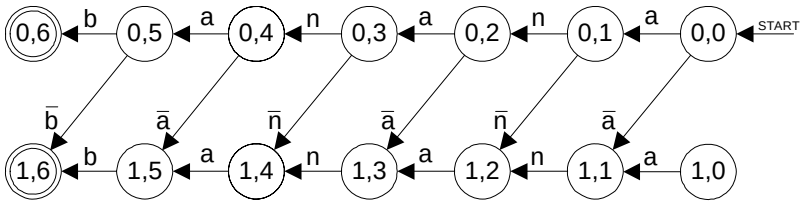


Figure 5: Transition diagram of base NFA for $\Theta = SFORCO$, $k = 1$ and $P_\Theta = \{banana\}$

Algorithm 4: Construction of BPMA from given NFA
Input: Nondeterministic finite automaton $M_{NFA} = (Q, A, \delta, q_0, F)$.
Output: Backward pattern matching automaton M_{BPMA}.
Method:

1 **if** $\exists q \in Q, \exists a \in A : q_0 \in \delta(q, a)$ **then**
2 $Q' \leftarrow Q \cup \{q_{00}\},\ q'_0 \leftarrow q_{00}$
3 $\delta'(q, a) \leftarrow \delta(q, a)$ for all $q \in Q, a \in A$
4 $\delta'(q_{00}, a) \leftarrow \delta(q_0, a)$ for all $a \in A$
5 **else**
6 $Q' \leftarrow Q,\ q'_0 \leftarrow q_0$
7 $\delta'(q, a) \leftarrow \delta(q, a)$ for all $q \in Q, a \in A$
8 **end if**
9 $M'_{NFA} \leftarrow (Q', A, \delta', q'_0, F)$
10
11 $\delta''(q', a) \leftarrow \delta'(q', a)$ for all $q' \in Q', a \in A$
12 $\delta''(q'_0, a) \leftarrow \bigcup_{q' \in Q' \setminus \{q'_0\}} \delta'(q', a)$ for all $a \in A$
13 $M''_{NFA} \leftarrow (Q', A, \delta'', q'_0, F)$
14
15 $t_{q,q',a}.ptf \leftarrow$ True for all $a \in A,\ q' \in \delta(q, a),\ q \in Q$
16 $t_{q'_0,q',a}.ptf \leftarrow$ True for all $a \in A,\ q' \in \delta'(q'_0, a)$
17 $t_{q'_0,q',a}.ptf \leftarrow$ False for all $a \in A,\ q' \in \bigcup_{q' \in Q' \setminus \{q'_0\}} \delta'(q', a)$
18 $M_{BPMA} \leftarrow (M''_{NFA}, R, \gamma_q, \gamma_\delta, G)$

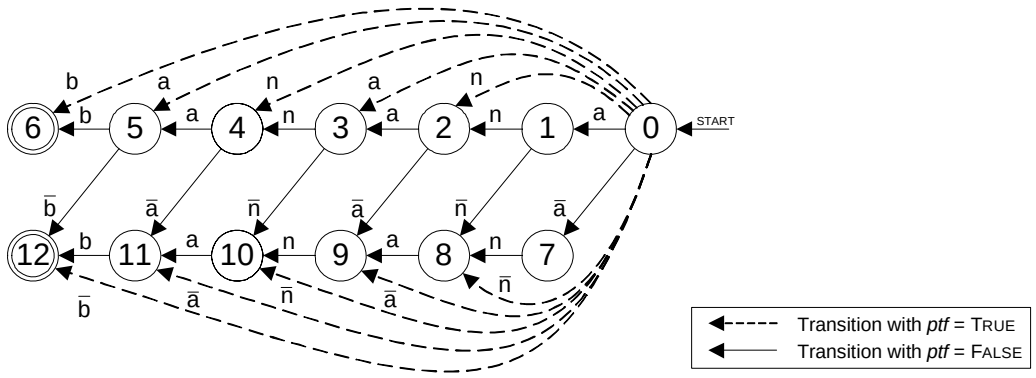


Figure 6: Transition diagram of model M of pattern matching problem $\Theta = SFORCO$ and pattern set $P_\Theta = \{banana\}$

We can now feed the resulting automaton M to Phase Two to perform the actual pattern matching. Let us take text $T =$*is it banana or ananas?* and run the Algorithm 1. The visualization of this process is presented in Figure 7.

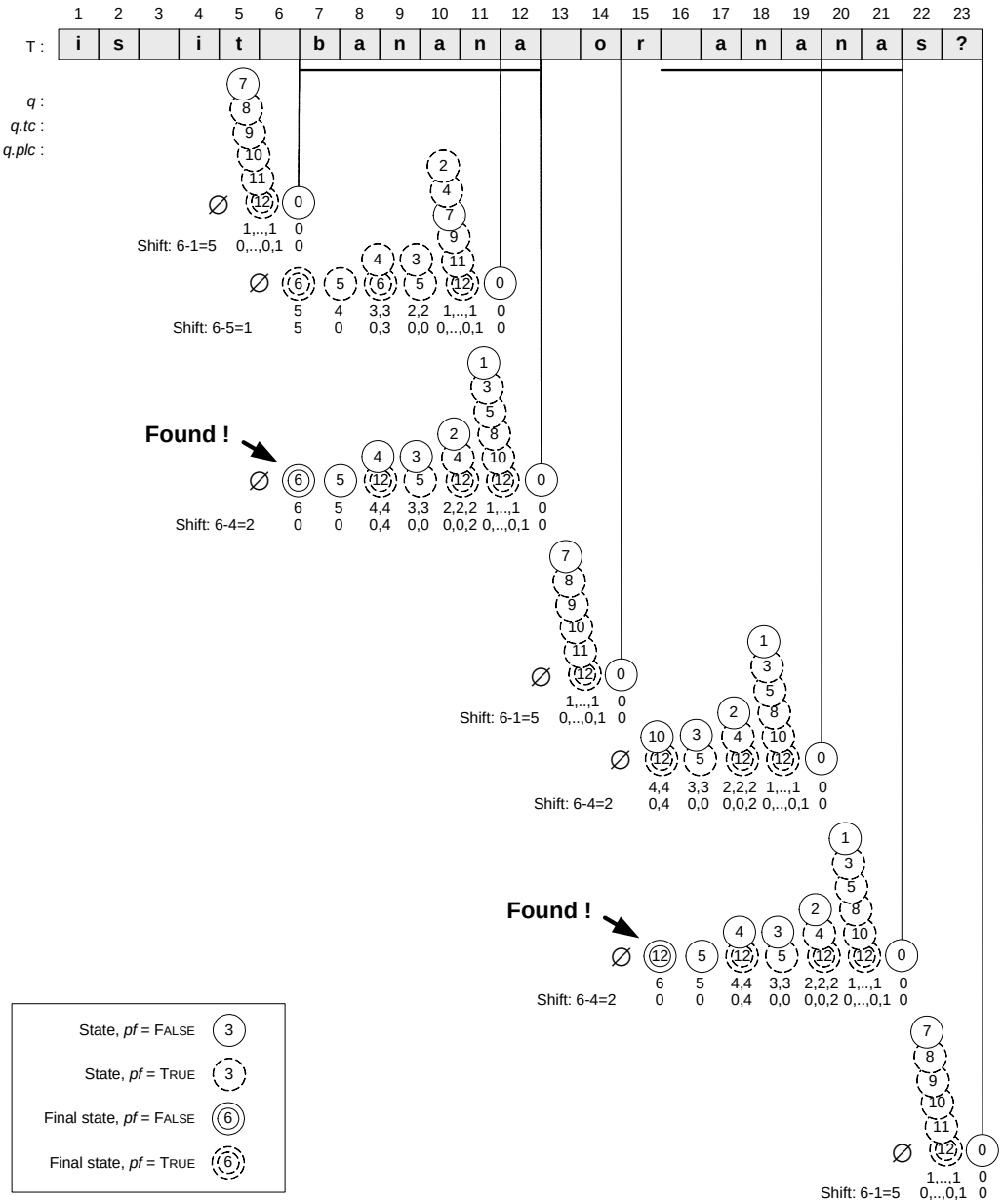


Figure 7: Pattern matching of $P = \{banana\}_{SFORCO}$ in text $T =$ *is it banana or ananas?* from the example

9 Future Work

The presented algorithm posses some drawbacks, that have to be solved in future work:

1. The finite automaton used to specify the pattern matching algorithm is nondeterministic and an equivalent deterministic automaton cannot be constructed by any known algorithm, because the automaton is attributed. To resolve this issue, a new algorithm constructing the equivalent attributed deterministic automaton has to be invented.

2. The longest safe shift computed by the current algorithm is not optimal. This shift can be further optimized by the observation mentioned at the end of Section 7.1 (Step 4).

3. The pattern matching algorithm presented in this report has the upper bound of its time complexity set higher than $O(n)$, where n is the length of text. The upper bound can be theoretically lowered to $O(n)$ but this optimization is yet to be found.

References

[BM77] R. S. Boyer, J. S. Moore: *A fast string searching algorithm.* C. ACM, Vol. 20, No. 10, pp. 762-772, October 1977.

[MH97] B. Melichar, J. Holub: *6D Classification of Pattern Matching Problems.* Proceedings of the Prague Stringology Club Workshop '97, July 1997, pp. 24-32.

[MH97a] B. Melichar, J. Holub: *Pattern Matching and Finite Automata.* In Proceedings of Summer School of Information Systems and Their Applications 1998, Ruprechtov, Czech Republic, September 1998, pp. 154-183.

[HAM50] R. W. Hamming: *Error-detecting and error-correcting codes.* Bell System Technical Journal 29:2, 1950, pp. 147-160.

[CR94] M. Crochemore, W. Rytter: *Text Algorithms.* Oxford University Press, 1994.

Bit-Parallel Computation of Local Similarity Score Matrices with Unitary Weights

Heikki Hyyrö[1] and Gonzalo Navarro[2*]

[1] Department of Computer Sciences, University of Tampere, Finland.
e-mail: heikki.hyyro@gmail.com

[2] Department of Computer Science, University of Chile.
e-mail: gnavarro@dcc.uchile.cl

Abstract. Local similarity computation between two sequences permits detecting all the relevant alignments present between subsequences thereof. A well-known dynamic programming algorithm works in time $O(mn)$, m and n being the lengths of the subsequences. The algorithm is rather slow when applied over many sequence pairs. In this paper we present the first bit-parallel computation of the score matrix, for a simplified choice of scores. If the computer word has w bits, then the resulting algorithm works in $O(mn \log \min(m, n, w)/w)$ time, achieving up to 8-fold speedups in practice. Some DNA comparison applications use precisely the simplified scores we handle, and thus our algorithm is directly applicable. In others, our method can be used as a raw filter to discard most of the strings, so the classical algorithm can be focused only on the substring pairs that can yield relevant results.

1 Introduction and Related Work

Sequence comparison is a fundamental task in Computational Biology, in order to detect relevant similarities between a pair of genetic or protein sequences [3]. Three kinds of similarities are of interest: (*i*) global similarity compares two strings as a whole, (*ii*) semiglobal (or semilocal) similarity looks for substrings of a given string that are similar to a second given string, (*iii*) local similarity looks for similar substrings of two given strings.

Similarity is usually expressed using a *score function*, which gives prizes or penalties to operations on the strings and to pairings of characters of the two strings. Usually pairing the same character in both strings involves a prize because we have found a similarity. Pairing different characters, inserting or removing characters, involves penalties. The specific values for prizes and penalties depend on the biological model used for the similarity (for example, logarithms of mutation probabilities). The similarity is then expressed as the highest possible score of a sequence of operations that align one string to the other.

Global and semiglobal similarity find applications in other areas such as text searching. Global similarity computation is then seen as a *distance computation*. The

*Partially funded by Millennium Nucleus Center for Web Research, Grant P04-067-F, Mideplan, Chile.

distance is never negative, and the smaller it is, the more similar the sequences are. Semiglobal similarity can be converted into an approximate search problem, namely to find the approximate occurrences of a short pattern inside a long text. Local similarity, on the other hand, is more specific to computational biology applications.

All these sorts of similarity computations can be easily carried out in $O(mn)$ time using dynamic programming. Given strings $A_{1...m}$ and $B_{1...n}$, the general method is to compute an $(m+1) \times (n+1)$ matrix C whose cell $C_{i,j}$ gives the maximum score/minimum distance to align/convert $A_{...i}$ to $B_{...j}$. The cells of row 0 and column 0 form initially known boundary cases, and the remaining $m \times n$ cells are computed using a recurrence. For example, for global similarity score computation we may have $C_{i,0} = -i$, $C_{0,j} = -j$, and for $i, j > 0$

$$\begin{aligned} C_{i,j} &= \max(C_{i-1,j-1} + \delta(A_i, B_j),\ C_{i,j-1} - 1,\ C_{i-1,j} - 1) \\ \text{where } \delta(A_i, B_j) &= \text{if } A_i = B_j \text{ then } 1 \text{ else } -1 \end{aligned}$$

where we have assumed that all penalties are -1 and prizes are $+1$. More complicated score functions can be real-valued and depend on the characters involved. The maximum score for the strings A and B is $C_{m,n}$.

If we are instead computing distance, we may have $C_{i,0} = i$, $C_{0,j} = j$, and for $i, j > 0$

$$\begin{aligned} C_{i,j} &= \min(C_{i-1,j-1} + \delta(A_i, B_j),\ C_{i,j-1} + 1,\ C_{i-1,j} + 1) \\ \text{where } \delta(A_i, B_j) &= \text{if } A_i = B_j \text{ then } 0 \text{ else } 1 \end{aligned}$$

where we have assumed that all costs are 1. The minimum distance between A and B is $C_{m,n}$.

Semiglobal similarity computation is obtained by using the above formulas except that $C_{0,j} = 0$, so that an alignment of A can start afresh at any position in B. High score/low distance at cell $C_{m,j}$ tells us that an interesting alignment ends at position j in B.

Local similarity computation needs a somewhat different arrangement and, curiously, it seems not expressible using the distance model, but just the score model. In this case we have $C_{i,0} = C_{0,j} = 0$, and for $i, j > 0$

$$\begin{aligned} C_{i,j} &= \max(0,\ C_{i-1,j-1} + \delta(A_i, B_j),\ C_{i,j-1} - 1,\ C_{i-1,j} - 1) \\ \text{where } \delta(A_i, B_j) &= \text{if } A_i = B_j \text{ then } 1 \text{ else } -1 \end{aligned}$$

where we remark the 0 value involved in the maximum. The objective of this zero is that if an alignment in progress has given us more penalties than prizes, then it is better to start afresh from that position. Any cell value $C_{i,j}$ that is high enough indicates that similar substrings end at position i in A and j in B.

Much effort has been carried out in order to efficiently compute the *distance* matrix, both for global and semiglobal alignments. In particular, *bit-parallelism* has given the best results in practice. Bit-parallelism packs several values inside a computer word and updates them all in one shot. The bit-parallel algorithm that best "parallelizes" the matrix computation is from Myers [8], which computes semiglobal similarity and is easily adapted to compute global similarity [4, 5, 6]. Using Myers' algorithm, both similarities can be computed in $O(mn/w)$ time using a computer

word of w bits, which is the optimal bit-parallel speedup. Myers' algorithm strongly relies on the fact that consecutive cells of $C_{i,j}$ differ only by -1, 0, or 1. Several other bit-parallel algorithms exploiting the same property have been proposed [9].

Other approaches to speed up the computation exist. Different Four-Russians techniques [7, 11] obtain $O(mn/\log(mn))$ time. The same complexity is obtained by using a Ziv-Lempel factoring [2], which generalizes to local similarity with arbitrary weights. In practice, when applicable, bit-parallel algorithms are faster.

Bit-parallel computation of *score* matrices, however, has not been attempted. Bergeron and Hamel [1] have extended Myers' scheme to handle arbitrary integer weights for substitutions, as well as a fixed weight c for insertions and deletions. Their algorithm is $O(mnc\log(c)/w)$ time. This scheme cannot be used to compute local similarity.

In general, global and semiglobal score computation can be converted into distance computation. However, local similarity is of different nature and cannot be easily mapped to a known distance computation scheme. In this paper we present a bit-parallel algorithm inspired on Myers' scheme (and more precisely on Hyyrö's version [4]), which obtains $O(mn\log\min(m,n,w)/w)$ time. The algorithm assumes that aligning two characters yields a prize of $+1$ when they are equal and a penalty of -1 otherwise, and that inserting or deleting characters has a penalty of -1.

The main obstacles to obtain the algorithm are (1) that the recurrence is more complicated than the one afforded by Myers (in particular, differences of $+2$ among contiguous cells are possible), and (2) that the zero in the maximization involves knowing absolute cell values, while the whole philosophy of Myers' scheme relies on storing differential values.

We implemented the algorithm and compared it against plain dynamic programming, which is currently the only alternative. We show that up to 8-fold speedups are obtained using our algorithm.

Our algorithm cannot replace dynamic programming because it cannot handle other prize and penalty values. On the other hand, while score computations on protein sequences are always weighted, there are many cases of score computations on DNA sequences where our simplified model is actually used [3]. It may also be feasible to use our method as a fast filter to discard most of the matrix and let the weighted dynamic programming algorithm concentrate only on the matrix areas that look promising.

2 A Bit-Parallel Design

Let us focus on the simple score function depicted in the Introduction, that is,

$$\begin{aligned} C_{i,0} &= C_{0,j} = 0 \quad \text{and, for } i,j>0, \\ C_{i,j} &= \max(0,\ C_{i-1,j-1}+\delta(A_i,B_j),\ C_{i,j-1}-1,\ C_{i-1,j}-1) \\ \text{where } \delta(A_i,B_j) &= \text{if } A_i = B_j \text{ then } 1 \text{ else } -1 \end{aligned}$$

We prove now some properties of matrix C. Note, to start, that C contains no negative values.

Lemma 1: Given the above definition of matrix C, it holds

$$\begin{aligned} C_{i,j} - C_{i-1,j-1} &\in -1, 0, +1 && \text{for any } i,j > 0 \\ C_{i,j} - C_{i,j-1} &\in -1, 0, +1, +2 && \text{for any } i \geq 0, j > 0 \\ C_{i,j} - C_{i-1,j} &\in -1, 0, +1, +2 && \text{for any } i > 0, j \geq 0 \end{aligned}$$

Proof: We proceed inductively, so we assume it proved for any (i', j') such that $j' < j$, or $j' = j$ and $i' < i$. The base cases are immediate. Now, for the inductive case, let us start with the first proposition. The option $C_{i-1,j-1} + \delta(A_i, B_j)$ in the "max" clause of the formula for $C_{i,j}$ guarantees that $C_{i,j} - C_{i-1,j-1} \geq -1$. Inductive Hypothesis tells us that $C_{i-1,j} \leq C_{i-1,j-1} + 2$ and $C_{i,j-1} \leq C_{i-1,j-1} + 2$, and thus $C_{i,j} = \max(0,\ C_{i-1,j-1} + \delta(A_i, B_j),\ C_{i,j-1} - 1,\ C_{i-1,j} - 1) \leq \max(C_{i-1,j-1} + \delta(A_i, B_j),\ C_{i-1,j-1} + 1,\ C_{i-1,j-1} + 1) = C_{i-1,j-1} + 1$. Here we removed the zero from the "max" clause as it is known that $C_{i-1,j-1} + 1 \geq 1 > 0$. By combining the two previous observations, we have that $-1 \leq C_{i,j} - C_{i-1,j-1} \leq 1$.

Let us now consider the second proposition. First we note that $C_{i,j} - C_{i,j-1} \geq -1$ because of the option $C_{i,j-1} - 1$ inside the "max" clause. From our Inductive Hypothesis and the above-proved first proposition we have that $C_{i,j-1} \geq C_{i-1,j-1} - 1 \geq C_{i,j} - 1 - 1 = C_{i,j} - 2$. Thus $-1 \leq C_{i,j} - C_{i,j-1} \leq 2$. The third proposition is symmetric with the second and comes out similarly. □

Given the ranges of values proved for consecutive differences, we will represent matrix C incrementally using the following *bit matrices*:

$$\begin{aligned} M_{i,j} &\equiv A_i = B_j & \qquad DP_{i,j} &\equiv C_{i,j} - C_{i-1,j-1} = +1 \\ Z_{i,j} &\equiv C_{i,j} = 0 & \qquad DZ_{i,j} &\equiv C_{i,j} - C_{i-1,j-1} = 0 \\ & & \qquad DM_{i,j} &\equiv C_{i,j} - C_{i-1,j-1} = -1 \\ HT_{i,j} &\equiv C_{i,j} - C_{i,j-1} = +2 & \qquad VT_{i,j} &\equiv C_{i,j} - C_{i-1,j} = +2 \\ HP_{i,j} &\equiv C_{i,j} - C_{i,j-1} = +1 & \qquad VP_{i,j} &\equiv C_{i,j} - C_{i-1,j} = +1 \\ HZ_{i,j} &\equiv C_{i,j} - C_{i,j-1} = 0 & \qquad VZ_{i,j} &\equiv C_{i,j} - C_{i-1,j} = 0 \\ HM_{i,j} &\equiv C_{i,j} - C_{i,j-1} = -1 & \qquad VM_{i,j} &\equiv C_{i,j} - C_{i-1,j} = -1 \end{aligned}$$

Here M and Z stand for "match" and "zero", respectively. D, H, and V stand for "diagonal", "horizontal", and "vertical", respectively. T, P, Z, and M stand for "plus two", "plus one", "zero", and "minus one", respectively. When a cell refers to a value out of bounds, such as $HP_{i,0}$, its value is not really important.

The above information clearly represents the cells of matrix C. For example,

$$C_{i,j} = \sum_{r=1}^{i} (2 \times VT_{r,j} + 1 \times VP_{r,j} - 1 \times VM_{r,j})$$

The next step is to derive logical properties that relate those bit matrices, so as to permit an efficient bit-parallel implementation.

$DP_{i,j} \equiv M_{i,j} \vee VT_{i,j-1} \vee HT_{i-1,j}$:
It is clear that if either $A_i = B_j$, $C_{i,j-1} = C_{i-1,j-1} + 2$, or $C_{i-1,j} = C_{i-1,j-1} + 2$, then $C_{i,j} = C_{i-1,j-1} + 1$. Moreover, if none of them hold, there is no way for $C_{i,j}$ to get the value $C_{i-1,j-1} + 1$.

$DZ_{i,j} \equiv \sim DP_{i,j} \wedge (Z_{i-1,j-1} \vee VP_{i,j-1} \vee HP_{i-1,j})$:

From the score recurrence we can easily derive the rule that $C_{i,j} = C_{i-1,j-1}$ if and only if $C_{i,j} \neq C_{i-1,j-1}+1$ and $\max(0, C_{i,j-1}-1, C_{i-1,j}-1) = C_{i-1,j-1}$. Moreover, since $0 \leq C_{i-1,j-1}$ and the condition $C_{i,j} \neq C_{i-1,j-1}+1$ implies that $C_{i,j-1} < C_{i-1,j-1}+2$ and $C_{i-1,j} < C_{i-1,j-1}+2$, it turns out that already $C_{i-1,j-1} \geq \max(0, C_{i,j-1}-1, C_{i-1,j}-1)$, so the condition $\max(0, C_{i,j-1}-1, C_{i-1,j}-1) = C_{i-1,j-1}$ can be changed into the form $C_{i-1,j-1} \in \{0,\ C_{i,j-1}-1,\ C_{i-1,j}-1\}$. This results in the above formula for $DZ_{i,j}$.

$DM_{i,j} \equiv \sim (DP_{i,j} \vee DZ_{i,j})$: As it is the only remaining choice.

$HT_{i,j} \equiv DP_{i,j} \wedge VM_{i,j-1}$:

From now on we build on $D*$ and the other bit matrices, by exhaustively examining all the choices for $C_{i,j} - C_{i-1,j-1}$ using submatrices where the lower right cell is $C_{i,j} = x$ and the upper left can thus have a value $x-1$, x or $x+1$. The lower left cell is $C_{i,j-1}$, which in this particular item must have the value $x-2$. We discard cases that are not possible according to Lemma 1 and express the remaining cases as logical conditions. We put "$\times$" in the remaining corner to signal impossible cases.

$x-1$	
$x-2$	x

x	$\times$
$x-2$	x

$x+1$	$\times$
$x-2$	x

$HP_{i,j} \equiv (DP_{i,j} \wedge VZ_{i,j-1}) \vee (DZ_{i,j} \wedge VM_{i,j-1})$:

$x-1$	
$x-1$	x

x	
$x-1$	x

$x+1$	$\times$
$x-1$	x

$HM_{i,j} \equiv VT_{i,j-1} \vee (DZ_{i,j} \wedge VP_{i,j-1}) \vee (DM_{i,j} \wedge VZ_{i,j-1})$:

$x-1$	
$x+1$	x

x	
$x+1$	x

$x+1$	
$x+1$	x

Note the simplification in the first condition since $VT_{i,j-1} \Rightarrow DP_{i,j}$.

$HZ_{i,j} \equiv \sim (HT_{i,j} \vee HP_{i,j} \vee HM_{i,j})$: As it is the only remaining choice.

$VT_{i,j} \equiv DP_{i,j} \wedge HM_{i-1,j}$:

Now we focus on the upper right corner.

$x-1$	$x-2$
	x

x	$x-2$
$\times$	x

$x+1$	$x-2$
$\times$	x

$VP_{i,j} \equiv (DP_{i,j} \wedge HZ_{i-1,j}) \vee (DZ_{i,j} \wedge HM_{i-1,j})$:

$x-1$	$x-1$
	x

x	$x-1$
	x

$x+1$	$x-1$
$\times$	x

$VM_{i,j} \equiv HT_{i-1,j} \vee (DZ_{i,j} \wedge HP_{i-1,j}) \vee (DM_{i,j} \wedge HZ_{i-1,j})$:

$x-1$	$x+1$
	x

x	$x+1$
	x

$x+1$	$x+1$
	x

$VZ_{i,j} \equiv \sim (VT_{i,j} \vee VP_{i,j} \vee VM_{i,j})$: As it is the only remaining choice.

3 A Bit-Parallel Algorithm

Up to now we have focused on how to compute the C matrix without regard for which should be the output of the algorithm. As explained, computational biologists are interested in matrix positions where the local score exceeds some threshold k. Those positions are then subject of further analysis.

Hence our algorithm will receive two strings A and B, as well as a threshold value k, and will point out all the positions (i, j) of matrix C where the score of the local alignment between $A_{...i}$ and $B_{...j}$ is at least k, that is, where $C_{i,j} \geq k$.

The idea of the bit-parallel algorithm is to process C column by column (just like the standard dynamic programming algorithm). However, the bit-parallel algorithm will process all the column in one shot, not row by row. In this section we assume $m \leq w$, that is, we can pack all bits of a column $G_j = G_{1...m,j}$ in a single computer word, for any matrix G. Note that row zero is not represented. When needed, the ith bit of vector G_j will be written as $G_j(i) = G_{i,j}$.

Therefore, our computation will proceed with column bit vectors DP_j, DM_j, DZ_j, and so on, for $j = 0 \ldots n$, each packed in a computer word. After step j of the algorithm, the vectors will hold the bits corresponding to column j of the matrix.

We will use the usual C instructions to handle bits: "&" as the bitwise-and, "|" as the bitwise-or, "^" as the bitwise-xor, "$\sim$" as the bitwise-not, and "<<" to shift all the bits one position to the left and enter a zero at the rightmost position. Sometimes we will treat bit vectors as integers and perform arithmetic operations on them.

In a precomputation step, explained in Section 3.1, the "match" matrix M is built in a suitable way for bit-parallel processing. The boundary conditions of matrix C are handled by giving the proper values to Z_0 and $V*_0$ vectors, namely $VP_0 = VM_0 = VT_0 = 0$ and $Z_0 = VZ_0 = 2^m - 1$. Then we process the characters of B (matrix columns) one by one. Each step j computes the bit vectors for column j from the vectors of column $j - 1$. First, the diagonal vectors $D*_j$ as well as the horizontal vector HP_j are computed. Vector HP_j is computed already at this stage as we use it in computing DZ_j. This part is complex and is explained in Section 3.2. Then the rest of the horizontal and vertical vectors $H*_j$ and $V*_j$ are easy to compute, as explained in Section 3.3. Finally, in Section 3.4, we show how to find and report high enough scores in column j, and how the same mechanism handles also computing vector Z_j. The way this last part is done is again slightly complicated and uses a technique that is rather different from all the rest.

3.1 Computing Matrix M

Matrix M is represented as a table indexed by alphabet characters. $M[c]$ is a bit vector such that $M[c](i) = 1$ iff $A_i = c$. This table is precomputed before filling matrix C. This way the cell value $M_{i,j}$ is actually represented by $M[B_j](i)$.

Matrix M is precomputed in $O(m + |\Sigma|)$ time, where Σ is the alphabet of A and B, as follows. First initialize $M[c] \leftarrow 0$ for every $c \in \Sigma$ and then traverse string A character-wise, setting bit $M[A_i](i) \leftarrow 1$.

3.2 Computing Vectors $D*_j$ and HP_j

Let us start with DP_j. As seen in Section 2, $DP_{i,j} \equiv M_{i,j} \vee VT_{i,j-1} \vee HT_{i-1,j}$. Since we are computing all the values at column j in one shot, component $HT_{i-1,j}$ is troublesome because it is not yet computed ($M_{i,j} = M[B_j](i)$ is known so it is not problematic). Let us expand $HT_{i-1,j}$ using its definition:

$$DP_{i,j} \equiv M_{i,j} \vee VT_{i,j-1} \vee (DP_{i-1,j} \wedge VM_{i-1,j-1})$$

where now the problematic value belongs to the same DP column. Let us express this recurrence in vector form. We define temporary vectors $X(i) \equiv M[B_j](i) \vee VT_{j-1}(i)$ and $Y(i) \equiv VM_{j-1}(i)$. Then the recurrence for vector DP_j is

$$DP_j(i) \equiv X(i) \vee (DP_j(i-1) \wedge Y(i-1))$$

This particular kind of circular dependency has already been solved by Myers [8] in his simpler formulation for edit distance computation. Following Hyyrö's explanation [4, 10], we unroll $DP_j(i-1)$ to obtain

$$DP_j(i) \equiv X(i) \vee (X(i-1) \wedge Y(i-1)) \vee (DP_j(i-2) \wedge Y(i-1) \wedge Y(i-2))$$

and unrolling repeatedly we obtain

$$DP_j(i) \equiv \vee_{r=0}^{i} \left(X(i-r) \wedge \left(\wedge_{s=i-r}^{i-1} Y(s)\right)\right)$$

that is, any bit set in X before position i can propagate through a sequence of bits set in Y that reach position $i-1$, so as to set position i in DP_j. Myers [8] has shown that the above formula can be computed using bit-parallelism as follows:

$X \leftarrow M[B_j] \mid VT_{j-1}$
$Y \leftarrow VM_{j-1}$
$DP_j \leftarrow ((Y + (X \,\&\, Y)) \,\hat{}\, Y) \mid X$

Let us now consider DZ. From Section 2 we have

$$DZ_{i,j} \equiv \sim DP_{i,j} \wedge (Z_{i-1,j-1} \vee VP_{i,j-1} \vee HP_{i-1,j})$$

where this time the problem arises with $HP_{i-1,j}$. But it turns out that vector HP_j can be computed once the vector DP_j is known. In Section 2 we gave the formula

$$HP_{i,j} \equiv (DP_{i,j} \wedge VZ_{i,j-1}) \vee (DZ_{i,j} \wedge VM_{i,j-1})$$

for it. If we look at the situation where the condition $DZ_{i,j} \wedge VM_{i,j-1}$ is true, we can have $C_{i,j} = x$ only if $C_{i-1,j} = x+1$, that is, only if $HP_{i-1,j}$ is true. Also, $DP_{i,j}$ must obviously be false. Hence, $DZ_{i,j} \wedge VM_{i,j-1} \Rightarrow HP_{i-1,j} \wedge VM_{i,j-1} \wedge \sim DP_{i,j}$. Moreover, it is straighforward to see that the condition $DZ_{i,j} \wedge VM_{i,j-1}$ is true whenever $HP_{i-1,j} \wedge VM_{i,j-1} \wedge \sim DP_{i,j}$ is true, and thus we have the following alternative formula for $HP_{i,j}$:

$$HP_{i,j} \equiv (DP_{i,j} \wedge VZ_{i,j-1}) \vee (HP_{i-1,j} \wedge VM_{i,j-1} \wedge \sim DP_{i,j})$$

The circular dependency on HP_j can be solved in similar fashion as in the case of computing vector DP_j. In this case, defining temporary vectors X and Y such that

$X(i) \equiv DP_j(i) \wedge VZ_{j-1}(i)$ and $Y(i) \equiv VM_{j-1}(i+1) \wedge \sim DP_j(i+1)$, the preceding formula for $HP_{i,j}$ gets the vector form

$$HP_j(i) \;\equiv\; X(i) \;\vee\; (HP_j(i-1) \;\wedge\; Y(i-1))$$

which is identical to the previous circular dependency for computing DP_j. We get immediately the following bit-parallel formula for computing HP_j:

$X \leftarrow DP_j \;\&\; VZ_{j-1}$
$Y \leftarrow (VM_{j-1} \;\&\; \sim DP_j) >> 1$
$HP_j \leftarrow ((Y + (X \;\&\; Y)) \,\hat{}\, Y) \mid X$

Once vector HP_j is available, computing the vector DZ_j becomes easy: a straightforward conversion of its formula leads into the following bit-parallel code.

$$DZ_j \;\leftarrow\; \sim DP_j \;\&\; (((Z_{j-1} << 1) \mid 1) \mid VP_{j-1} \mid (HP_j << 1))$$

where, after the shift of Z_{j-1} we have introduced a "1" at its lowest bit to reflect the fact that $C_{0,j-1} = 0$ (that is, $Z_{0,j-1} = 1$) for any j (recall that row zero of Z is not represented). Similarly, $HP_{0,j} = 0$ because $C_{0,j} - C_{0,j-1} = 0 \neq 1$, so we leave the new rightmost bit in zero after shifting HP_j. Finally, we have the following simple bit-parallel formula for DM_j.

$$DM_j \;\leftarrow\; \sim (DP_j \mid DZ_j)$$

3.3 Computing Other Vectors $H*_j$ and $V*_j$

Once DP_j, HP_j, DM_j, and DZ_j corresponding to the current column j are computed, the rest flows easily by following the formulas used in Section 2. Again, when we shift a bit vector to the left, we add or not a "1" bit at the rightmost position depending on which is the value of that vector at the unrepresented row zero.

$HT_j \leftarrow DP_j \;\&\; VM_{j-1}$
$HM_j \leftarrow VT_{j-1} \mid (DZ_j \;\&\; VP_{j-1}) \mid (DM_j \;\&\; VZ_{j-1})$
$HZ_j \leftarrow \sim (HT_j \mid HP_j \mid HM_j)$
$VT_j \leftarrow DP_j \;\&\; (HM_j << 1)$
$VP_j \leftarrow (DP_j \;\&\; ((HZ_j << 1) \mid 1)) \mid (DZ_j \;\&\; (HM_j << 1))$
$VM_j \leftarrow (HT_j << 1) \mid (DZ_j \;\&\; (HP_j << 1)) \mid (DM_j \;\&\; ((HZ_j << 1) \mid 1))$
$VZ_j \leftarrow \sim (VT_j \mid VP_j \mid VM_j)$

3.4 Keeping Scores and Computing Vector Z_j

Once the bit vectors for column j have been computed, we check whether some cell values in column j of matrix C exceed the matching threshold k. At the same time it is also convenient to check which cells have the value zero and record those positions into vector Z_j. Unfortunately the differential information of the bit vectors does not allow us to make this in any simple and fast way. The naive approach would be to use the difference information between adjacent cell values to compute and check the cell values $C_{1\ldots m,j}$. This would take $O(m)$ time per column, making the overall run time $O(mn)$, the same as with classical dynamic programming.

On the other hand, as shown by Myers [8], a single value $C_{i,1\ldots n}$ can be tracked in constant time per column by using the horizontal vectors $H*_j$. The problem is that we need to track all the rows i, falling again to $O(m)$ time per column.

Our approach is to set up multiple *witnesses* into a single bit vector, and then scan the column in parallel with the witnesses. Each witness will be associated with some i and keep track of the cell values $C_{i,1\ldots n}$, that is, the cell values on row i of C. A somewhat similar method was used in [5, 6] as part of an approximate string matching algorithm.

Let MW_j be a length-m bit vector that holds the multiple witnesses at column j and let Q denote the number of bits taken by each witness. Then MW_j can hold $r = \lfloor m/Q \rfloor$ witnesses. Let $MW_j\{i\}$ denote a witness that has its first bit in position i of MW_j. $MW_j\{i\}$ occupies the bits $MW_j(i \ldots i+Q-1)$ and keeps track of the cell values on row i of C. The first witness is always $MW_j\{1\}$, and the rest are spread evenly into MW_j. This can be done in such manner that the largest empty gap after the region of any witness is $\lceil (m-rQ)/r \rceil$. Let us define $Q' = Q + \lceil (m-rQ)/r \rceil$, that is, Q' gives the maximum distance between the first bit of a witness and the first bit of the next witness or, for the last witness, the position after the last bit of the whole vector.

Assume that $C_{i,j} = x$ and the witness $MW_j\{i\}$ exists. For reasons that become clear below, we record the value x into $MW_j\{i\}$ in the form $2^{Q-1} - x$. To guarantee that the witnesses can represent all possible score values from zero to $\min(m, n)$, the parameter Q is determined as the minimum number for which $2^{Q-1} \geq \min(m, n)$, that is, $Q = \lceil \log_2 \min(m, n) \rceil + 1$. Figure 1 exemplifies (vectors S, E, K will be introduced soon).

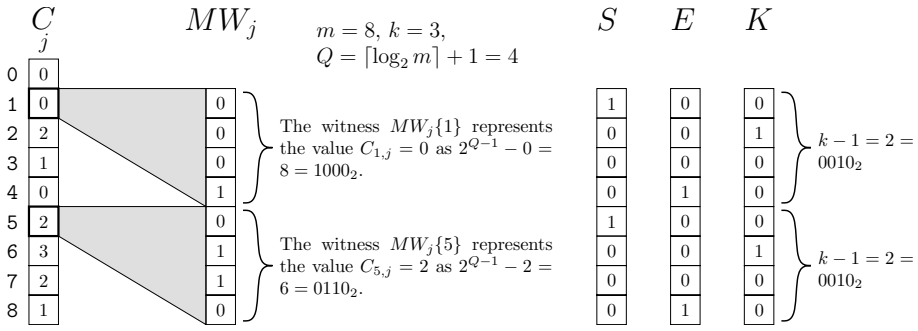


Figure 1: Example of usage of MW, S, E, and K vectors.

With these conventions the witnesses have the following properties:

(1) The Qth bit of $MW_j\{i\}$ is set if and only if $C_{i,j} = 0$.

(2) Adding some value x to $C_{i,j}$ corresponds to subtracting x from $MW_j\{i\}$, and vice versa.

(3) If we add $k-1$ to $MW_j\{i\}$, then the Qth bit of $MW_j\{i\}$ is set if and only if $C_{i,j} < k$.

The witnesses are initialized to $MW_0\{i\} = 2^{Q-1}$ since all values in column 0 of C are zero. After that the witness values are computed by using the horizontal vectors. For

example, if $MW_{j-1}\{i\} = x$ and the ith bit of HT_j is set, then $MW_j\{i\} = MW_{j-1}\{i\} - 2 = x - 2$ (note that we subtracted the $+2$ due to property (2)). When MW_{j-1} and the horizontal vectors $H*_j$ are available, all witnesses $MW_j\{1\} \ldots MW_j\{r\}$ may be computed in bit-parallel fashion. To achieve this, we use a "start" bit mask S with bits set in those locations that correspond to the first bits of witnesses. Then, the whole witness vector MW_j may be computed as:

$$MW_j \leftarrow MW_{j-1} - 2(HT_j \,\&\, S) - (HP_j \,\&\, S) + (HM_j \,\&\, S)$$

Once MW_j and the vertical vectors $V*_j$ are available, all cell values in column j of C can be scanned in bit-parallel manner. First we copy MW_j into an auxiliary vector X. At this stage each witness $MW_j\{i\}$ copied into X represents the value $C_{i,j}$. Then each witness MW_j is updated $Q' - 1$ times. First to represent the value $C_{i+1,j}$, then the value $C_{i+2,j}$, and so on until the value $C_{i+Q'-1,j}$. After $Q' - 1$ iterations, all cells of column j have been scanned (some possibly twice if $Q' \neq Q$). At each stage of the scan we check the current witness values for matches or zeros. For this we use an "end" bit mask $E \leftarrow S << (Q-1)$ that has a bit set in those positions that correspond to the last bits of the witnesses. In addition we use a bit mask K that holds the value $k - 1$ at each witness location.

When the witnesses $MW_j\{i\}$ in X represent the cells $C_{i+h,j}$, the vector $((X + K) \,\&\, E) >> (Q - 1 - h)$ has bits set in those positions u where $C_{u,j} < k$, and the vector $(X \,\&\, E) >> (Q - 1 - h)$ has bits set in those positions u where $C_{u,j} = 0$.

Our strategy for checking matches is to record during the scan whether column j contains any matches or not. These may then be checked more carefully, if desired, but if all matching locations are recorded exactly, the run time becomes again $O(mn)$ in the worst case.

The match checking is done by using an auxiliary vector Y that is initialized by setting $Y \leftarrow E$. When $MW_j\{i\}$ represents $C_{i+h,j}$, we set $Y \leftarrow Y \,\&\, (X + K)$. There is at least one match in column j if and only if $Y \neq E$ after the Q' iterations (consisting of the initial stage and $Q' - 1$ update stages). The zero vector Z_j is computed by initializing it to zero and setting $Z \leftarrow Z \mid ((X \,\&\, E) >> (Q - 1 - h))$ when $MW_j\{i\}$ represents $C_{i+h,j}$.

computes MW_j, then it updates the witnesses in the auxiliary vector X to go through all cell values in column j. It also records matching columns as well as computes the vector Z_j during the scan. The first stage is handled separately, and for this reason for example the vector Z_j is directly given a non-zero value.

Figure 2 gives the complete algorithm. Note that, by carefully choosing the update order, we manage to keep only one copy of each vector.

3.5 Analysis

Up to now we have assumed that $m \leq w$. In this case computing the M table takes $O(m+|\Sigma|)$ time, and the rest of the algorithm in Figure 2 clearly runs in time $O(nQ')$. Since $Q' < 2Q$ and $Q = \lceil \log_2 \min(m, n) \rceil + 1$, we have that $nQ' = O(n \log \min(m, n))$ and the total running time is $O(|\Sigma| + m + n \log \min(m, n))$.

If $m > w$, the length-m bit vectors can be simulated in $O(\lceil m/w \rceil)$ time by using $\lceil m/w \rceil$ vectors of length w (details are omitted for lack of space). This results in the time $O(m + \lceil m/w \rceil |\Sigma|)$ for computing the M table, and the run time of the rest of the

LocalScores $(A_{1\ldots m},\ B_{1\ldots n},\ k)$

1. **For** $c \in \Sigma$ **Do** $M[c] \leftarrow 0$
2. **For** $i \in 1 \ldots m$ **Do** $M[A_i] \leftarrow M[A_i] \mid 2^{i-1}$
3. $VP, VM, VT \leftarrow 0,\ VZ, Z \leftarrow 2^m - 1$
4. $Q \leftarrow \lceil \log_2 m \rceil + 1$
5. $r \leftarrow \lfloor m/Q \rfloor$
6. $S \leftarrow$ distribute evenly r witnesses and mark their first bit
7. $MW, E \leftarrow S << (Q-1)$
8. $K \leftarrow S \times (k-1)$
9. $Q' \leftarrow Q + \lceil (m - rQ)/r \rceil$
10. **For** $j \in 1 \ldots n$ **Do**
11. $\quad X \leftarrow M[B_j] \mid VT$
12. $\quad DP \leftarrow ((VM + (X \ \&\ VM)) \ \hat{}\ VM) \mid X$
13. $\quad X \leftarrow DP \ \&\ VZ$
14. $\quad Y \leftarrow (VM \ \&\ \sim DP) >> 1$
15. $\quad HP \leftarrow ((Y + (X \ \&\ Y)) \ \hat{}\ Y) \mid X$
16. $\quad DZ \leftarrow \sim DP \ \&\ (((Z << 1) \mid 1) \mid VP \mid (HP << 1))$
17. $\quad DM \leftarrow \sim (DP \mid DZ)$
18. $\quad HT \leftarrow DP \ \&\ VM$
19. $\quad HM \leftarrow VT \mid (DZ \ \&\ VP) \mid (DM \ \&\ VZ)$
20. $\quad HZ \leftarrow \sim (HT \mid HP \mid HM)$
21. $\quad VT \leftarrow DP \ \&\ (HM << 1)$
22. $\quad VP \leftarrow (DP \ \&\ ((HZ << 1) \mid 1)) \mid (DZ \ \&\ (HM << 1))$
23. $\quad VM \leftarrow (HT << 1) \mid (DZ \ \&\ (HP << 1)) \mid (DM \ \&\ ((HZ << 1) \mid 1))$
24. $\quad VZ \leftarrow \sim (VT \mid VP \mid VM)$
25. $\quad MW \leftarrow MW - 2(HT \ \&\ S) - (HP \ \&\ S) + (HM \ \&\ S)$
26. $\quad X \leftarrow MW$
27. $\quad Y \leftarrow E$
28. $\quad Z \leftarrow 0$
29. $\quad$ **For** $h \in 0 \ldots Q' - 1$ **Do**
30. $\quad\quad Z \leftarrow Z \mid ((X \ \&\ E) >> (Q - 1 - h))$
31. $\quad\quad Y \leftarrow Y \ \&\ (X + K)$
32. $\quad\quad X \leftarrow X - 2((VT_j >> h) \ \&\ S) - ((VP_j >> h) \ \&\ S) + ((VM_j >> h) \ \&\ S)$
33. $\quad$ **If** $Y \neq E$ **Then** `Record match at column` j

Figure 2: Complete bit-parallel algorithm to compute local similarity. Some optimizations have been discarded for clarity.

algorithm is multiplied by a factor of $O(\lceil m/w \rceil)$, which yields $O(mn \log \min(m,n)/w)$, taking the alphabet size as a constant for simplicity.

The run time $O(mn \log \min(m,n,w)/w)$ mentioned in the beginning of the paper is finally achieved by observing that the values in different length-w segments of the bit vectors may be stored using delta encoding. If $C_{\lceil hw+w/2 \rceil, j} = x$ for some $h \geq 0$, we know from Lemma 1 that the values in the corresponding length-w section, $C_{hw+1 \ldots (h+1)w, j}$, must be in the range $x - w + 1 \ldots x + w$. Thus if the witnesses for section $C_{hw+1 \ldots (h+1)w, j}$ represent the values of form $C_{\lceil hw+w/2 \rceil, j} - C_{i,j}$, we may use the value $Q = \lceil \log_2 \min(m, n, 2w) \rceil + 1$ in storing the witnesses. Here we use the value $2w$ instead of w in order to ensure that the sum $X + K$ in match checking does not cause an overflow. Note that this scheme requires some modifications in the process of checking for zero values and/or matches. For example the values in K must be adjusted depending on the current base-value x.

Compared to the best bit-parallel complexity for global and semiglobal similarity (actually, for distance computation), $O(mn/w)$, we have a logarithmic penalty factor because of the use of local similarity. At this point it should be clear that we can compute global and semiglobal scores (rather than distances) within the same $O(mn/w)$ complexity, just by removing the use of vector Z and checking the score only at a single cell or a single row. This removes the need for the witnesses and their logarithmic penalty.

4 Experimental Results

We implemented the $O(mn \log \min(m,n,w)/w)$ version of our algorithm and compared it to the plain dynamic programming algorithm. Both algorithms were programmed in C, and we tried to make both implementations as efficient as possible. The test computer was a 64-bit Sparc Ultra 2 with 128 MB RAM, and the codes were compiled with GCC 3.3.1 with optimization switched on. The test strings were randomly selected DNA sequences from the genome of S. cerevisiae (baker's yeast). The test contained two different types of scenarios. In the first we tested with short patterns and a long text. This test involved the matching thresholds $k = 1$ and $k = m-1$ to see what kind of effect the value of k has. In the second we tested aligning patterns and texts that have the same length, and this time we used only $k = m - 1$. The results are shown in Fig. 3 (left and right, respectively). In them our algorithm is observed to be 1.2 - 8.5 times faster than the basic dynamic programming algorithm when $w = 64$.

5 Conclusions

We have presented the first bit-parallel algorithm to compute local similarity score between two strings, which has many practical applications in computational biology. While dynamic programming, the only existing algorithm, takes time $O(mn)$ (m and n being the lengths of the strings), our algorithm needs time $O(mn \log \min(m,n,w)/w)$ using a computer word of w bits. Our experiments show up to 8-fold speedups.

Our algorithm cannot replace dynamic programming because it cannot handle prize and penalty values other than ± 1. However, it can be used as a fast filter

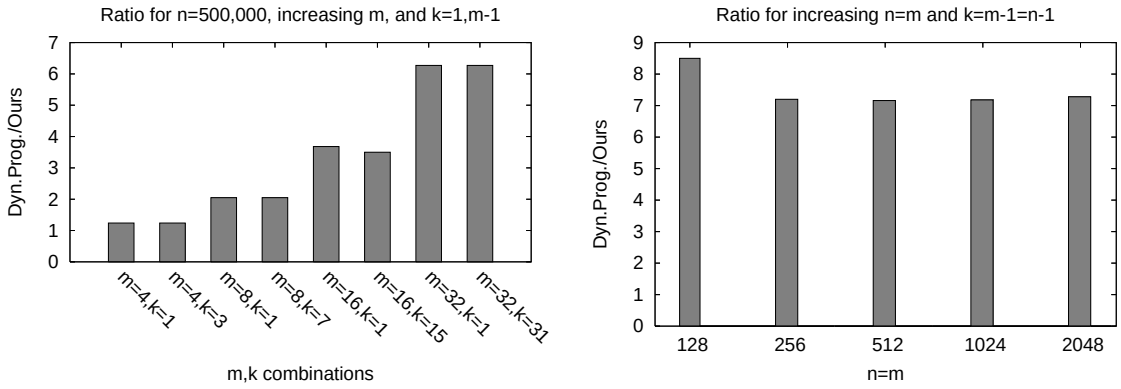


Figure 3: Speedup factor of our bit-parallel algorithm over the basic dynamic programming algorithm. On the left, aligning long against short strings. On the right, aligning strings of the same length.

to discard most of the areas and let the dynamic programming algorithm concentrate only on the areas that look promising. Moreover, there are some DNA-related applications where they use precisely those ± 1 penalties.

As future research issues, the most immediate is to investigate whether it is possible to "pack" the logical conditions describing the differences across the diverse directions in a different way that makes the overall formula faster to compute. Longer-term goals are accomodating other cost functions apart from the unitary-cost one, and trying to obtain optimal speedup, removing the term $O(\log \min(m, n, w))$ from the cost formula.

References

[1] A. Bergeron and S. Hamel. Vector algorithms for approximate string matching. *International Journal of Foundations of Computer Science*, 13(1):53–65, 2002.

[2] M. Crochemore, G. Landau, and M. Ziv-Ukelson. A sub-quadratic sequence alignment algorithm for unrestricted scoring matrices. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'02)*, pages 679–688, 2002.

[3] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[4] H. Hyyrö. Explaining and extending the bit-parallel approximate string matching algorithm of Myers. Technical Report A-2001-10, Dept. of Computer and Information Sciences, University of Tampere, Tampere, Finland, 2001.

[5] H. Hyyrö and G. Navarro. Faster bit-parallel approximate string matching. In *Proc. 13th Combinatorial Pattern Matching (CPM'02)*, LNCS 2373, pages 203–224, 2002.

[6] H. Hyyrö and G. Navarro. Bit-parallel witnesses and their applications to approximate string matching. *Algorithmica*, 41(3):203–231, 2005.

[7] W. Masek and M. Paterson. A faster algorithm for computing string edit distances. *J. of Computer and System Sciences*, 20:18–31, 1980.

[8] G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic progamming. *Journal of the ACM*, 46(3):395–415, 1999.

[9] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

[10] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings – Practical online search algorithms for texts and biological sequences.* Cambridge University Press, 2002.

[11] S. Wu, U. Manber, and E. Myers. A sub-quadratic algorithm for approximate limited expression matching. *Algorithmica*, 15(1):50–67, 1996.

A Space Efficient Bit-Parallel Algorithm for the Multiple String Matching Problem

Domenico Cantone and Simone Faro

Dipartimento di Matematica e Informatica, Università di Catania, Italy

e-mail: {cantone, faro}@dmi.unict.it

Abstract. Finite (nondeterministic) automata are very useful building blocks in the field of string matching. This is particularly true in the case of multiple pattern matching, where the use of factor-based automata can reduce substantially the number of computational steps when the patterns have large common factors.
Direct simulation of nondeterministic automata can be performed very efficiently using the bit-parallelism technique, though this is not necessarily true for factor-based automata.

In this paper we present an algorithm for the multiple string matching problem, based on the bit-parallel simulation of nondeterministic factor-based automata which satisfy a particular ordering condition. We also show how to enforce such condition by suitably modifying a minimal initial automaton, through equivalence preserving transformations. The resulting automaton turns out to be smaller than the corresponding maximal automata used by existing bit-parallel algorithms, as they do not take any advantage of common factors in patterns.

Keywords: multiple string matching, bit-parallelism, text searching.

1 Introduction

Given a set $\mathcal{P} = \{P_1, \ldots, P_r\}$ of patterns and a text T, all strings over a finite alphabet Σ of size σ, the *multiple pattern matching problem* is to determine all the positions where any of the patterns in $\mathcal{P}$ occurs in T. This problem arises naturally in many applications, and several algorithms exist to solve it. For example, the UNIX `fgrep` and `egrep` programs support multi-pattern matching through the `-f` option. The worst case complexity of multiple pattern matching is $\Omega(n)$ and it has been achieved by the Aho-Corasick algorithm [AC75]. From a practical point of view, the best average complexity bound for multi-pattern matching algorithms is $\mathcal{O}(n \log_\sigma(rm)/m)$, where m is the minimum length of any pattern in $\mathcal{P}$. Such bound has been reached, for instance, by the Dawg-Match algorithm [CCG+93] and by the Multi-BDM algorithm [CR94]. We cite also that the Boyer-Moore strategy has been extended to multi-pattern matching, such as in the Commenz-Walter [CW79] and in the Wu-Manber [WM91] algorithms.

In this paper we are mainly interested on automata based solutions of the pattern matching problem, and on their implementation by bit-parallelism. In general, (nondeterministic) automata allow to handle classes of characters and multiple patterns in a simple, efficient, and flexible way, leading to algorithms which are asymptotically optimal both in space and time [KMP77, AC75].

The bit-parallelism technique [BYG92] consists in exploiting the intrinsic parallelism of the bit operations inside a computer word. It can be profitably used for the simulation of finite automata even in their nondeterministic form.

The paper is organized as follows. After introducing in Section 2 the basic notations used in the paper, in Section 3 we survey the most significant algorithms for the single and multiple pattern matching problem which make use of factor-based deterministic finite automata. Then, in Section 4 we describe the bit-parallelism technique and discuss some of the single and multi-pattern matching algorithms based on it. Existing algorithms in the multi-pattern case do not take any particular advantage of the presence of large common factors in the patterns. Thus, in Section 5 we present a new solution for the multi-pattern matching problem which efficiently mixes the advantages in space obtained from factor-based automata with the simplicity and flexibility of bit-parallelism. Finally, we draw our conclusions and propose some hints for future work in Section 6.

2 Basic Definitions and Terminology

We introduce here the basic notations and terminology used in the paper. A string P of length m is represented as an array $P[0\,..\,m-1]$. Thus $P[i]$ will denote the $(i+1)$-st character of P, for $i = 0, \ldots, m-1$. We denote the length of P by $|P|$. In addition, if $\mathcal{P} = \{P_1, P_2, \ldots, P_r\}$ is a set of strings, we denote by $size(\mathcal{P})$ the sum of the lengths of its strings, namely $size(\mathcal{P}) = \sum_{i=1}^{r} |P_i|$.

For any two strings P and P', we write $P' \sqsupset P$ to indicate that P' is a *proper* suffix of P, $P' \sqsubset P$ to indicate that P' is a *proper* prefix of P, and $P.P'$ to denote the concatenation of P' to P. Given a set of patterns $\mathcal{P} = \{P_1, P_2, \ldots, P_r\}$ in an alphabet Σ, the *trie* $\mathcal{T}$ associated with $\mathcal{P}$ is a rooted directed tree, whose edges are labeled by single characters of Σ, such that (i) distinct edges out of a same node are labeled by distinct characters, (ii) all paths in $\mathcal{T}$ from the root are labeled by prefixes of the strings in $\mathcal{P}$, (iii) for each string P in $\mathcal{P}$ there exists a path in $\mathcal{T}$ from the root which is labeled by P.

If we do not insist on property (i) above, we obtain a more relaxed form of trie, which we call *nondeterministic trie*. Since all tries considered in this paper are nondeterministic, for the sake of simplicity we will refer to them just as "tries."

For any node p in a trie $\mathcal{T}$, we denote by $lbl(p)$ the string which labels the path from the root of $\mathcal{T}$ to the node p and put $len(p) = |lbl(p)|$, i.e., $len(p)$ is the length of the path from the root of $\mathcal{T}$ to p. Additionally, for any edge (p, q) in $\mathcal{T}$, we denote the label of (p, q) by $lbl(p, q)$. We also denote by $children_{\mathcal{T}}(p)$ the set of the children of p in the trie $\mathcal{T}$.

Given a (nondeterministic) trie $\mathcal{T}$ relative to a set of patterns $\mathcal{P} = \{P_1, P_2, \ldots, P_r\}$ over an alphabet Σ, we can naturally associate with $\mathcal{T}$ the following *canonical nondeterministic finite automaton (NFA)* $\widehat{\mathcal{T}} = (Q_{\mathcal{T}}, q_0, F_{\mathcal{T}}, \delta_{\mathcal{T}})$, where:

- $Q_{\mathcal{T}}$ is the set of nodes of $\mathcal{T}$ (*set of states*);
- $q_0 \in Q_{\mathcal{T}}$ is the root of $\mathcal{T}$ (*initial state*);
- $F_{\mathcal{T}} =_{Def} \{q \in Q_{\mathcal{T}} \mid lbl(q) \in \mathcal{P}\}$ (*set of final or terminal states*);
- $\delta_{\mathcal{T}} : Q_{\mathcal{T}} \times \Sigma \to \mathscr{P}(Q_{\mathcal{T}})$, with

$$\delta_{\mathcal{T}}(q,c) =_{Def} \begin{cases} \{p \in Q_{\mathcal{T}} \mid lbl(q).c = lbl(p)\} & \text{if } q \neq q_0 \\ \{p \in Q_{\mathcal{T}} \mid lbl(q).c = lbl(p)\} \cup \{q_0\} & \text{if } q = q_0\,, \end{cases}$$

  for $q \in Q_{\mathcal{T}}$, $c \in \Sigma$, and where $\mathscr{P}(\cdot)$ is the powerset operator (*transition function*).

Thus the words *node* and *state* will often be used interchangeably. Likewise, we will often identify a trie $\mathcal{T}$ with its corresponding NFA $\widehat{\mathcal{T}}$.

3 Automata Based String Matching Algorithms

Automata play a very important role in the design of efficient pattern matching algorithms. For instance the well known Knuth-Morris-Pratt algorithm [KMP77] uses a deterministic automaton that searches a pattern in a text by performing its transitions on the text characters. The main result relative to the Knuth-Morris-Pratt algorithm is that its automaton can be constructed in $\mathcal{O}(m)$-time and -space, whereas pattern search takes $\mathcal{O}(n)$-time, thus reaching the best bound for a pattern matching algorithm (as usual, m and n denote the length of the pattern and text, respectively). In the case of multiple pattern matching, the Aho-Corasick algorithm [AC75] has been the first having a linear behavior. It is also based on the automata approach and can be viewed much as a generalization of the Knuth-Morris-Pratt algorithm to the multi-pattern case. In particular, the Aho-Corasick automaton is a trie $\mathcal{T}$ for the set of patterns $\mathcal{P}$, with a failure function $f : Q_{\mathcal{T}} \to Q_{\mathcal{T}}$ which is followed when no transition is possible on a text character. The function f is defined on each node $u \in Q_{\mathcal{T}}$ in such way that:

- $lbl(f(u)) \sqsupset lbl(u)$, and
- $len(f(u)) \geq len(p)$, for each $p \in Q_{\mathcal{T}}$ such that $lbl(p) \sqsupset lbl(u)$.

The Aho-Corasick automaton can be constructed in linear time and space [CR94].

Automata based solutions have been also developed to design algorithms which have optimal sublinear performances on average. For instance, several algorithms have been developed to extend to the multiple pattern matching case the efficient Boyer-Moore strategy [BM77]. Among them, we cite the Commenz-Walter algorithm [CW79] which extends the Horspool algorithm [Hor80] through a suffix based approach. The Commenz-Walter algorithm starts by reading the text backwards from position j, initially set to $\ell = \min\{|P_k| \ : \ P_k \in \mathcal{P}\}$. Then characters are matched against the labels of the trie $\mathcal{T}$ for the set $\mathcal{P}^r$ of the reverse patterns. When a final state is reached, an occurrence is reported. If no matching is possible with the current character, then position j is shifted by the minimum nonnull depth in $\mathcal{T}$

of an edge labeled by the previous read character $T[j]$. If no edge in $\mathcal{T}$ is labeled by $T[j]$, then j is increased by ℓ.

Another type of automaton, called *suffix automaton* (or DAWG, for Directed Acyclic Word Graph), has been introduced for the single pattern matching problem in [CCG+93, CCG+94, CR94, Raf97] and later generalized to the multi-pattern case. A suffix automaton for a set $\mathcal{P}$ of patterns is a trie for the set $\mathcal{P}^r$ that recognizes all the suffixes of the patterns in $\mathcal{P}$.

For instance, the REVERSE-FACTOR algorithm [CCG+94], for the single pattern matching problem, computes shifts which match prefixes of the pattern, rather than suffixes, using the smallest suffix automaton of the reverse of the pattern. Despite its quadratic worst-case time complexity, the REVERSE-FACTOR algorithm is very fast in practice. Other optimal sublinear algorithms on average, like BACKWARD-DAWG-MATCH (BDM) and TURBO-BDM [CCG+94, CR94], have been obtained with this approach, and have been also extended to multiple pattern matching in [CCG+93, CR94, Raf97].

4 String Matching and Bit-Parallelism

In general, it is much easier to construct a nondeterministic automaton rather than a deterministic one, due to its simplicity and regularity. Thus, it would be desirable to be able to simulate efficiently the parallel computation of an NFA. This can be done using the bit-parallelism technique [BYG92]. Such technique consists in exploiting the intrinsic parallelism of the bit operations inside a computer word. In favorable cases it allows to cut down the overall number of operations by a factor of ω, where ω is the number of bits in a computer word. For this reason, although string matching algorithms based on bit-parallelism are usually simple and have very low memory requirements, they generally work well only with patterns of moderate length.

In the context of string matching, such technique has been especially used to speed-up algorithms based on automata. The simulation is carried out by representing an automaton as an array of L bits, where $L+1$ is the number of states of the automaton. The initial state does not need to be represented, because it is always active. Bits corresponding to active states are set to `1`, whereas bits corresponding to inactive states are set to `0`.

To simulate efficiently an NFA using the bit-parallelism technique, its states must be mapped into the positions of a bit-vector by a suitable bijection.

In the case of a trie (or better, the NFA associated with it), we succeded to simulate it efficiently provided that the bijection is a *weakly safe topological ordering*, in a sense which will be explained later.

For the time being, we just recall that a topological ordering of a trie $\mathcal{T}$ is a bijection $\pi : Q_{\mathcal{T}} \to \{0, \ldots, |Q_{\mathcal{T}}| - 1\}$, which agrees with the edges of $\mathcal{T}$, namely such that $\pi(p) < \pi(q)$ whenever (p, q) is in $\mathcal{T}$. It is convenient to associate with π its inverse $\phi : \{0, \ldots, |Q_{\mathcal{T}}| - 1\} \to Q_{\mathcal{T}}$, which is assumed to map each position of a bit-vector to the corresponding state of $\mathcal{T}$.

For later purposes, given a topological ordering π of $\mathcal{T}$, it is also convenient to associate to each edge (p, q) in $\mathcal{T}$ its π-*interval* $[\pi(p), \pi(q)[$, also denoted by $Int_\pi(p, q)$. The length $\pi(q) - \pi(p)$ of the π-interval $[\pi(p), \pi(q)[$ will be denoted by $|Int_\pi(p, q)|$.

Notice that since π is a topological ordering of $\mathcal{T}$, then $|Int_\pi(p,q)| \geq 1$, for each edge (p,q) in $\mathcal{T}$.

4.1 Searching for a Single Pattern

In the case of single pattern matching, the trie $\mathcal{T}$ associated with a given pattern P of length m is linear. Thus, the corresponding NFA $\widehat{\mathcal{T}}$ is obtained from $\mathcal{T}$ just by adding a self-loop on its initial state, labeled by all symbols of the alphabet Σ, to allow the scan to begin at any position in the text. Plainly, in this case we have only one possible topological ordering of $\mathcal{T}$, whose inverse ϕ_1 is recursively defined by:

$$\phi_1(i) = \begin{cases} \delta_{\mathcal{T}}(q_0, P[0]) & \text{if } i = 0 \\ \delta_{\mathcal{T}}(\phi_1(i-1), P[i-1]) & \text{if } 1 \leq i \leq m-1\,. \end{cases}$$

Thus, for $i = 0, 1, \ldots, m-1$, state $\phi_1(i)$ is simulated by the i-th bit of a bit-vector. The initial state does not need to be represented, because it is always active. Figure 1(A) shows the nondeterministic finite automaton which recognizes the pattern $P = \texttt{aababb}$.

The first result, concerning single pattern matching algorithms using the bit-parallelism technique, is due to Baeza-Yates and Gonnet [BYG92]. Their algorithm, named Shift-And, maintains, for each symbol c of the alphabet Σ, a bit mask $B[c]$ whose i-th bit is set to 1, provided that $P[i] = c$, where P is the pattern. The current configuration of the automaton is maintained in a bit mask D, which is initialized to $\texttt{0}^L$, since initially all (noninitial) states are inactive. Moreover a *final-state* bit-mask $M = \texttt{10}^{L-1}$ maintains the position of the final state of the automaton, whereas an *initial-state* bit-mask $I = \texttt{0}^{L-1}\texttt{1}$ maintains the position of the node adjacent to the initial state.

While scanning a text T from left to right, the Shift-And algorithm simulates automaton transitions by the following basic shift-and operation, for each position j:

$$D = ((D \ll 1) \mid I) \,\&\, B[T[j]]\,.$$

If the final state is active, i.e. $D \,\&\, M \neq \texttt{0}^L$, a matching is reported at position j. It turns out that the Shift-And algorithm has an $\mathcal{O}(\lceil mn/\omega \rceil)$ worst-case running time and requires $\mathcal{O}(\lceil L/\omega \rceil)$-space.

Other algorithms based on bit-parallelism use a Boyer-Moore strategy, to simulate a right to left scan of the pattern. For instance, the BNDM algorithm is the bit-parallel implementation of the Reverse-Factor algorithm. It is based on the nondeterministic version of the smallest suffix automaton of the reverse of the pattern P. Unlike the Shift-And algorithm, characters of text and pattern are compared from right to left until the entire pattern is read or no transition by the automaton is possible. Then the pattern is shifted by ℓ positions to the right, where ℓ is the length of the last matched prefix. Despite its quadratic worst-case running time, the BNDM algorithm performs well in practical cases.

4.2 Searching for Multiple Patterns

Existing algorithms that search for a set $\mathcal{P} = \{P_1, \ldots, P_r\}$ of patterns, using bit-parallelism, simulate the behavior of the *maximal trie* of $\mathcal{P}$. This is the trie $\mathcal{T}$ of

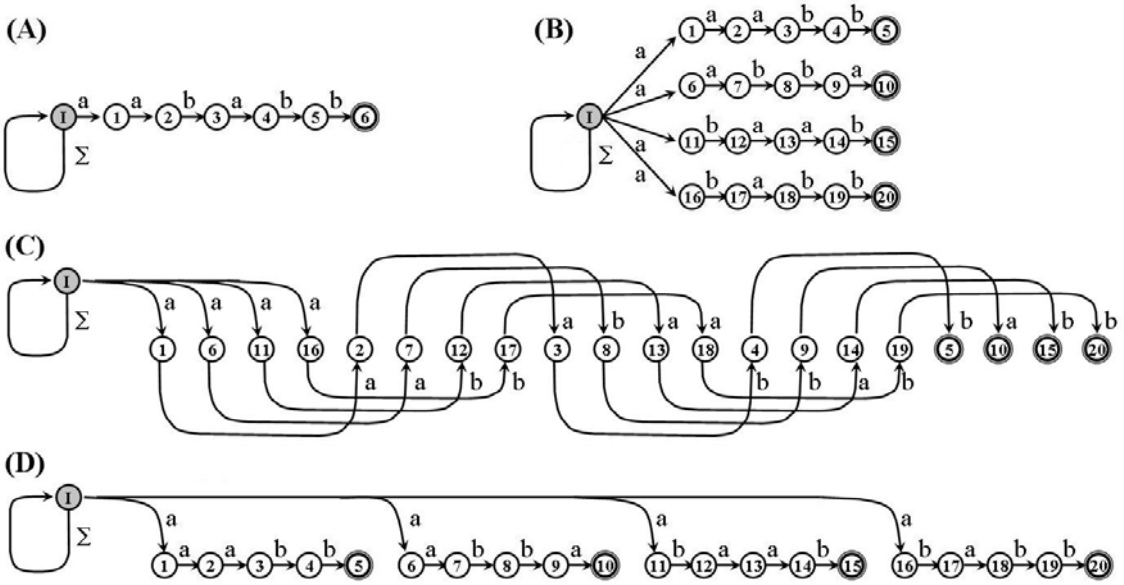


Figure 1: **(A)** An NFA which recognizes the pattern P = aababb. **(B)** An NFA obtained from the maximal trie $\mathcal{T}$ of the set of patterns $\mathcal{P}$ = {aaabb, aabba, abaab, ababb}. **(C)** The parallel topological ordering of $\mathcal{T}$. **(D)** The sequential topological ordering of $\mathcal{T}$.

$\mathcal{P}$ obtained from the linear tries $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_r$ for the patterns $P_1, P_2, \ldots, P_r$, respectively, by merging the roots of $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_r$ in a single node. Plainly, the number of states of $\mathcal{T}$ is given by $|\mathcal{T}| = \sum_{i=1}^{r} |\mathcal{T}_i| - r + 1 = size(\mathcal{P}) + 1$, so that it can be represented by a bit-vector of $L = size(\mathcal{P})$ bits. For instance, Figure 1(B) shows the maximal trie relative to the set of patterns $\mathcal{P}$ = {aaabb, aabba, abaab, ababb}. Two different topological orderings have been used in literature to simulate a maximal trie of a set of pattern $\mathcal{P}$. A first arrangement, π^{par}, has been proposed in [WM91], under the restriction that all patterns in the set $\mathcal{P}$ have the same length. Given a set $\mathcal{P} = \{P_1, P_2, \ldots, P_r\}$ of r distinct patterns of the same length m, the topological ordering π^{par} of the trie $\mathcal{T}$ relative to $\mathcal{P}$ is obtained just by interleaving the NFAs of the patterns of $\mathcal{P}$ in a parallel fashion. More precisely, the inverse ϕ^{par} of π^{par} can be recursively defined by

$$\phi^{\text{par}}(kr + j) = \begin{cases} \delta_{\mathcal{T}_{j+1}}(q_0, P_{j+1}[0]) & \text{if } k = 0 \\ \delta_{\mathcal{T}_{j+1}}(\phi^{\text{par}}((k-1)r + j), P_{j+1}[k]) & \text{if } 1 \leq k \leq m-1\,, \end{cases}$$

with $0 \leq j \leq r - 1$. Figure 1(C) shows the parallel topological ordering of the NFA of Figure 1(B). Using such arrangement, it is possible to search for patterns in $\mathcal{P}$ just as in the case of a single pattern. The only difference with the single pattern case is that the shift is not by a single bit, but by r bits (since consecutive nodes are r bits apart in the parallel arrangement). Moreover, we need to use the new initial-state and final-state masks $I = \mathtt{0}^{r(m-1)}\mathtt{1}^r$ and $M = \mathtt{1}^r\mathtt{0}^{r(m-1)}$, respectively. Figure 2 (left side) shows the code of an implementation of the Shift-And algorithm, based on a parallel ordering of the maximal trie for a set $\mathcal{P}$ of patterns having the same length.

An alternative arrangement, π^{seq}, has been proposed in [NR98]. It consists in

```
PARALLEL-SHIFT-AND (T, {P_1, ..., P_r})
1.  n = length(T)
2.  m = length(P_1)
3.  L = m^r
4.  for c ∈ Σ do B[c] = 0^L
5.  l = 0
6.  for i = 0 to m − 1 do
7.    for k = 1 to r do
8.      B[P_k[i]] = (B[P_k[i]] | (0^{L−1}1 ≪ l + k))
9.    l = l + r
10. I = 0^{r(m−1)}1^r
11. M = 1^r 0^{r(m−1)}
12. D = 0^L
13. for j = 0 to n − 1 do
14.   if D & M ≠ 0^L then print(j)
15.   D = ((D ≪ r) | I) & B[T[j]]
```

```
SEQUENTIAL-SHIFT-AND (T, {P_1, ..., P_r})
1.  n = length(T)
2.  m = length(P_1)
3.  L = m^r
4.  for c ∈ Σ do B[c] = 0^L
5.  l = 0
6.  for k = 1 to r do
7.    for i = 0 to m − 1 do
8.      B[P_k[i]] = (B[P_k[i]] | (0^{L−1}1 ≪ l + i))
9.    l = l + m
10. I = (0^{m−1}1)^r
11. M = (10^{m−1})^r
12. D = 0^L
13. for j = 0 to n − 1 do
14.   if D & M ≠ 0^L then print(j)
15.   D = ((D ≪ 1) | I) & B[T[j]]
```

Figure 2: On the left, the PARALLEL-SHIFT-AND algorithm which uses a parallel ordering of the maximal trie $\mathcal{T}$ of the set $\mathcal{P}$, and, on the right, the SEQUENTIAL-SHIFT-AND algorithm which uses a sequential ordering of the nodes of $\mathcal{T}$.

concatenating in a sequential fashion the different branches of the maximal trie of a set $\mathcal{P}$ of patterns. More precisely, given a set $\mathcal{P} = \{P_1, P_2, \ldots, P_r\}$ of patterns (not necessarily of the same length), the inverse ϕ^{seq} of the ordering π^{seq} relative to the maximal trie of $\mathcal{P}$ is recursively defined by

$$\phi^{\text{seq}}\left(\sum_{j=1}^{h-1}|P_j|+i\right) = \begin{cases} \delta_{\mathcal{T}_h}(q_0, P_h[0]) & \text{if } i = 0 \\ \delta_{\mathcal{T}_h}(\phi^{\text{seq}}(\sum_{j=1}^{h-1}|P_j|+i-1), P_h[i-1]) & \text{if } 1 \le i \le |P_h|-1\,, \end{cases}$$

with $1 \le h \le r$.

Figure 1(D) shows the sequential topological ordering of the NFA in Figure 1(B). In this case, we return to single bit shifts, whereas the initial-state and final-state masks are

$$\begin{aligned} I &= (0^{|P_1|-1}1)(0^{|P_2|-1}1)\cdots(0^{|P_r|-1}1) \\ M &= (10^{|P_1|-1})(10^{|P_2|-1})\cdots(10^{|P_r|-1})\,. \end{aligned}$$

On some processors, shifts by a single position is faster than shift by $r > 1$ positions. In such cases the arrangement π^{seq} yields faster algorithms. Moreover, as already observed, such arrangement allows to deal with sets of patterns of different lengths.

Figure 2 (right side) shows the code of an implementation of the SHIFT-AND algorithm, based on a sequential ordering of the maximal trie of a set $\mathcal{P}$. Though not necessary, for the sake of simplicity we have assumed that the patterns in $\mathcal{P}$ have the same length m.

5 A new space efficient approach

In this section we propose a new approach to bit-parallel multiple pattern matching. Unlike existing solutions, presented in the previous section, which make use of the

maximal trie of a set $\mathcal{P}$ of patterns, here we propose a solution which simulates, using bit-parallelism, a factor-based automaton thus reducing the number of states and, accordingly, the number of bits needed for its representation.

Below we introduce the important notion of *(weakly) safe topological ordering* of a trie. Then, in Section 5.1 we present an efficient variant of the SHIFT-AND algorithm, based on a trie for $\mathcal{P}$ admitting a weakly safe topological ordering. Our proposed algorithm, called MULTIPLE-TRIE-SHIFT-AND, searches a text T for any pattern in a set $\mathcal{P}$ in $\mathcal{O}(n\lceil L/\omega\rceil)$-time, where $n = |T|$, $L = size(\mathcal{P})$, and ω is the size of a computer word. Subsequently, in Section 5.2 we present an algorithm, named CONSTRUCT-SAFE-TOPOLOGICAL-ORDERING, which given a (minimal) trie $\mathcal{T}$ for a set $\mathcal{P}$ of patterns constructs another trie $\mathcal{T}'$ for $\mathcal{P}$ admitting a weakly safe topological ordering (in general, the size of $\mathcal{T}'$ may be larger than the size of $\mathcal{T}$). The CONSTRUCT-SAFE-TOPOLOGICAL-ORDERING algorithm is based on a DFS approach and runs in $\mathcal{O}(L)$-time and -space, under suitable hypotheses.

Let π_u be a topological ordering of the subtrie $\mathcal{T}_u$ of $\mathcal{T}$ rooted in u. Also, let (p, q) be an edge of $\mathcal{T}_u$.

We say that (p, q) is a *long-bit edge (relative to the ordering π_u)* if the length of the π_u-interval of (p, q) is greater than 1, i.e., in symbols, $|Int_{\pi_u}(p, q)| > 1$.[1] Otherwise, i.e. if $|Int_{\pi_u}(p, q)| = 1$, we say that (p, q) is a *1-bit edge (relative to the ordering π_u)*. Additionally, if (p, q) is a long-bit edge of $\mathcal{T}_u$, we say that the label $lbl(p, q)$ of the edge (p, q) is an *engaged symbol* for the node u. It is convenient to define the following function and set

$$\begin{aligned} \mathcal{L}_{\pi_u}(c) &=_{Def} \{(p, q) \in \mathcal{T}_u \mid lbl(p, q) = c \text{ and } |Int_{\pi_u}(p, q)| > 1\} \\ \mathcal{A}_{\pi_u} &=_{Def} \{c \in \Sigma \mid \mathcal{L}_{\pi_u}(c) \neq \emptyset\}, \end{aligned}$$

for c in the alphabet Σ, u in $\mathcal{T}$, and π_u a topological ordering of $\mathcal{T}_u$. In other words, $\mathcal{L}_{\pi_u}(c)$ is the collection of long-bit edges of $\mathcal{T}_u$ labeled by c, whereas $\mathcal{A}_{\pi_u}$ is the collection of all engaged symbols for u.

Finally, a topological ordering π of a trie $\mathcal{T}$ is said to be

- *safe*, if for each $c \in \Sigma$, the intervals in $\{Int_\pi(p, q) \mid (p, q) \in \mathcal{L}_\pi(c)\}$ are pairwise disjoint, i.e., if the π-intervals of any two distinct long-bit egdes labeled by a same character are disjoint;
- *weakly safe*, if for each $c \in \Sigma$, the intervals in $\{Int_\pi(p, q) \mid (p, q) \in \mathcal{L}_\pi(c)$ and $p \neq root(\mathcal{T})\}$ are pairwise disjoint, i.e., if the π-intervals of any two distinct long-bit egdes labeled by a same character and not originating from the root of $\mathcal{T}$ are disjoint.

Figures 3(B)-(C) show two different topological orderings of the trie in Figure 3(A). In particular, concerning the ordering π' relative to Figure 3(B), we have $\mathcal{L}_{\pi'}(a) = \{(3, 6), (8, 9)\}$ and $\mathcal{L}_{\pi'}(b) = \{(1, 2)\}$; hence π' is a weakly safe topological ordering since $\pi'(9) = 6 < 10 = \pi'(3)$. On the other hand, the ordering π'' relative to Figure 3(C) is not weakly safe, since in this case we have $\mathcal{L}_{\pi''}(a) = \{(1, 8), (3, 6), (8, 9)\}$, $\mathcal{L}_{\pi'}(b) = \emptyset$, and $\pi''(1) = 1 < \pi''(3) = 3 < \pi''(6) = 6 < \pi''(8) = 8$, i.e. $Int_{\pi''}(3, 6) \subset Int_{\pi''}(1, 8)$.

[1] The notion of π_u-interval and the notation $|Int_{\pi_u}(p, q)|$ have been introduced just before Section 4.1.

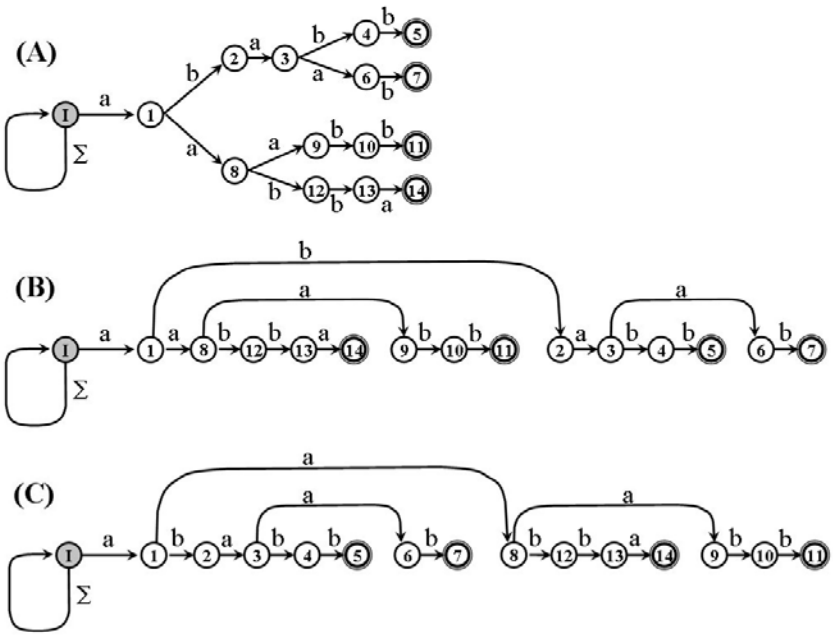


Figure 3: **(A)** The minimal trie of the set of patterns $\mathcal{P} = \{\texttt{ababb}, \texttt{abaab}, \texttt{aaabb}, \texttt{aabba}\}$. **(B)** A weakly safe topological ordering of the trie in (A). **(C)** A topological ordering of the trie in (A) which is not weakly safe.

5.1 The Multiple-Trie-Shift-And Algorithm

Given a text T and a set $\mathcal{P} = \{P_1, P_2, \ldots, P_r\}$ of patterns, the Multiple-Trie-Shift-And algorithm which we present below searches for any pattern of $\mathcal{P}$ in the text T in $\mathcal{O}(n\lceil L/\omega \rceil)$-time, where $n = |T|$, $L = \mathit{size}(\mathcal{P})$, and ω is the size of a computer word. Besides the text T, it takes as input a pair $\mathcal{T}$ and π, where $\mathcal{T}$ is a trie for $\mathcal{P}$ and π is a weakly safe topological ordering of $\mathcal{T}$ (as will be shown in the next section, such $\mathcal{T}$ and π can be efficiently constructed starting from a minimal trie for $\mathcal{P}$). The Multiple-Trie-Shift-And algorithm simulates its input automaton $\mathcal{T}$ using bit-parallelism. Since $|Q_{\mathcal{T}}| \leq L + 1$, in general our algorithm deals with smaller automata than the algorithms reviewed in Section 4.2.

Let $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_h$ be the subtries of $\mathcal{T}$ rooted in the children of $\mathit{root}(\mathcal{T})$ and let $\{f_1, f_2, \ldots, f_k\}$ be the set of final states of $\mathcal{T}$. The algorithm initializes two bit-masks of length $L = |\mathcal{T}| - 1$, respectively the initial-state mask I and the final-state mask M, as follows

$$\begin{aligned} I &= (0^{|Q_{\mathcal{T}_h}|-1}1)\cdots(0^{|Q_{\mathcal{T}_2}|-1}1)(0^{|Q_{\mathcal{T}_1}|-1}1) \\ M &= (10^{\pi(f_k)-\pi(f_{k-1})-1})\cdots(10^{\pi(f_2)-\pi(f_1)-1})(10^{\pi(f_1)-1})\,. \end{aligned}$$

Subsequently, for each symbol $c \in \Sigma$, the Multiple-Trie-Shift-And algorithm initializes as shown below three more bit-masks of length L, namely $B[c], IS[c]$ and $GS[c]$, which allow to perform the automaton transitions.

For each state $q \in Q_{\mathcal{T}}$ such that $\mathit{lbl}(q)[\mathit{len}(q) - 1] = c$, we set the $\pi(q)$-th bit of $B[c]$ to 1.

Let $\mathcal{L}_\pi(c) = \{(p_1, q_1), (p_2, q_2), \ldots, (p_t, q_t)\}$ be the set of long-bit edges in π labeled

```
MULTIPLE-TRIE-SHIFT-AND (T, 𝒯, π)
   /* INITIALIZATION */
 1.  n = length(T)
 2.  φ = π^{-1}
 3.  L = |Q_𝒯| − 1
 4.  I = M = 0^L
 5.  for each c ∈ Σ do B[c] = IS[c] = GS[c] = 0^L
 6.  root = φ(0)
 7.  for each q ∈ children_𝒯(root) do
 8.      c = lbl(root, q)
 9.      B[c] = (B[c] | (0^{L−1}1 ≪ (π(q) − 1)))
10.  for i = 1 to L do
11.      p = φ(i)
12.      if IS_FINAL(p) then M = (M | (0^{L−1}1 ≪ (i − 1)))
13.      if p ∈ children_𝒯(root) then I = (I | (0^{L−1}1 ≪ (i − 1)))
14.      for each q ∈ children_𝒯(p) do
15.          c = lbl(p, q)
16.          if π(q) > i + 1 then
17.              IS[c] = (IS[c] | (0^{L−1}1 ≪ (π(q) − 1)))
18.              GS[c] = (GS[c] | (0^{L−π(q)+π(p)+1}1^{π(q)−π(p)−1} ≪ π(p)))
19.          else B[c] = (B[c] | (0^{L−1}1 ≪ (π(q) − 1)))
   /* SEARCHING PHASE */
20.  D = 0^L
21.  for j = 0 to n − 1 do
22.      if D & M ≠ 0^L then print(j)
23.      D' = (D ≪ 1) & B[T[j]]
24.      D'' = ((((D & IS[T[j]]) ≪ 1) + GS[T[j]]) & ∼GS[T[j]])
25.      D = (D' | D'') | (I & B[T[j]])
```

Figure 4: The MULTIPLE-TRIE-SHIFT-AND algorithm for the multiple string matching problem.

by the symbol c, arranged in such a way that $\pi(p_1) < \pi(q_1) \leq \pi(p_2) < \pi(q_2) \leq \cdots \leq \pi(p_t) < \pi(q_t)$. The mask $IS[c]$ is the *initial-shift* bit-mask of c. It marks all nodes in π from which a long-bit edge labeled with symbol c originates. In other words, for each edge $(p, q) \in \mathcal{L}_\pi(c)$, the $\pi(p)$-th bit of $IS[c]$ is set to 1. More formally,

$$IS[c] = (0^{L-p_t}1)(0^{p_t-p_{t-1}-1}1)\cdots(0^{p_2-p_1-1}1)(0^{p_1-1})\,.$$

Finally, the mask $GS[c]$ is the *gap-shift* bit-mask of c. For each long-bit edge $(p, q) \in \mathcal{L}_\pi(c)$, the bits of $GS[c]$ from position $(\pi(p)+1)$ up to position $(\pi(q)-1)$ are set to 1. More formally,

$$GS[c] = (0^{L-q_t+1}1^{q_t-p_t-1})(0^{p_t-q_{t-1}+1}1^{q_{t-1}-p_{t-1}-1})\cdots(0^{p_2-q_1+1}1^{q_1-p_1-1})(0^{p_1})\,.$$

During the searching phase (lines 20-25), a bit-mask D maintains the active state of the automaton. For each position j of the text T, the algorithm performs three main steps

1-bit transitions (line 22):

This is made in a simple way by shifting the mask D by one position to the

left. Then all transitions labeled with symbols different from $T[j]$ are deleted by performing an AND operation with the bit-mask $B[T[j]]$. More formally, the operation that simulates 1-bit transitions is

$$(D \ll 1) \,\&\, B[T[j]]\,.$$

Long-bit transitions (line 23):
First, the operation $(D \,\&\, IS[T[j]])$ isolates all active states from which long-bit edges originate. Then the resulting mask is shifted by one position to the left and its value is added to the value of the bit-mask $GS[T[j]]$. This has the effect that, if $(p, q) \in \mathcal{L}_\pi(T[j])$ and p is an active state in D, then the $\pi(q)$-th bit of D is set to `1` and all bits from position $\pi(p)$ up to position $\pi(q) - 1$ are set to `0`. However, if $(p, q) \in \mathcal{L}_\pi(T[j])$ and p is not an active state in D, then all bits from position $\pi(p) + 1$ up to position $\pi(q) - 1$ maintain their value `1`. These undesirable bits are deleted by performing an AND operation with the bit-mask $\sim GS[T[j]]$. More formally, long-bit transitions are simulated by the operation

$$((((D \,\&\, IS[T[j]]) \ll 1) + GS[T[j]]) \,\&\, \sim GS[T[j]])\,.$$

Transitions from the initial state (line 24):
The transitions starting from the initial state are performed by computing an OR operation with the mask I. As in the 1-bit transition case, all transitions labeled with symbols different from $T[j]$ are deleted by performing an AND operation with the bit-mask $B[T[j]]$. Formally, transitions from the initial state are simulated by the following operation

$$(D \mid I) \,\&\, B[T[j]]\,.$$

The Multiple-Trie-Shift-And algorithm, shown in Figure 4, runs in $\mathcal{O}(n)$ time if $L \leq \omega$, where ω is the length of a computer word. However if $L > \omega$ the algorithm has a $\mathcal{O}(n\lceil L/\omega \rceil)$ worst-case time complexity.

In the following section we describe an algorithm that, given a minimal trie $\mathcal{T}$ for a set $\mathcal{P} = \{P_1, P_2, \ldots, P_r\}$ of patterns, it constructs another trie $\mathcal{T}'$, equivalent to $\mathcal{T}$, together with a weakly safe topological ordering π for $\mathcal{T}'$.

5.2 Constructing a Trie with a Weakly Safe Topological Ordering

Before entering into the details of the algorithm, we need to introduce some further useful concepts.

For each node $q \in Q_\mathcal{T}$ we define the set $B(q)$ of *binding symbols* of q as the collection of all characters which label some edge (p, p') originating from a predecessor p of q, but such that p' does not lie on the path from the $root(\mathcal{T})$ to q. In symbols

$$B(q) =_{Def} \{lbl(p, p') \mid p, p' \in Q_\mathcal{T}\,,\ lbl(p) \sqsubset lbl(q)\,,\ \text{and } lbl(p') \not\sqsubseteq lbl(q)\}\,.$$

In addition, for each node $q \in Q_{\mathcal{T}}$, we define the function $bind_q : \Sigma \to \{1, 2, \ldots, len(q)\}$ such that for each $c \in \Sigma$

$$bind_q(c) =_{Def} \begin{cases} 1 + \max \left\{ len(p) \;\middle|\; \begin{array}{l} p \in Q_{\mathcal{T}}\,,\ lbl(p) \sqsubset lbl(q)\,, \text{ and} \\ c = lbl(p, p')\,,\ lbl(p') \not\sqsubseteq lbl(q) \\ \text{for some } p' \in Q_{\mathcal{T}} \end{array} \right\} & \text{if } c \in B(q) \\ 0 & \text{otherwise}\,. \end{cases}$$

Observe that, if $lbl(p) \sqsubset lbl(q)$, then $len(p) < len(q)$ and therefore $0 \leq bind_q(c) \leq len(q)$, for $c \in \Sigma$. For each $h \in \{1, \ldots, len(q)\}$ we define the set $B_h(q) \subseteq B(q)$ by putting

$$B_h(q) =_{Def} \{c \in B(q) \mid bind_q(c) = h\}.$$

Next, let again $q \in Q_{\mathcal{T}}$ and let $w = |children_{\mathcal{T}}(q)|$. Also, for each node $s \in children_{\mathcal{T}}(q)$, let π_s be a safe topological ordering for $\mathcal{T}_s$. We say that the set $children_{\mathcal{T}}(q)$ is *resolved w.r.t. the above orderings* π_s, if there exists an ordering $s_1, s_2, \ldots, s_w$ of the children of q in $\mathcal{T}$ such that the concatenation $\pi_{s_1}.\pi_{s_2}.\cdots.\pi_{s_w}$ yields a safe topological ordering π_q for $\mathcal{T}_q$. Observe that the edge (q, s_1) is a 1-bit edge for π_q, whereas the edges (q, s_i), for $i = 2, \ldots, w$, are long-bit edges for π_q.

Then, in order for π_q to be a safe topological ordering, we must have

$$lbl(q, s_i) \notin \bigcup_{j=1}^{i-1} \mathcal{A}_{\pi_q}(s_j)\,, \qquad \text{for each } i = 1, \ldots, w\,.$$

Additionally, observe that the set $B_{len(q)}(s) = \{lbl(q, s') \mid s' \in children_{\mathcal{T}}(q) \setminus \{s\}\}$ defines the binding symbols on node s imposed by its predecessor q, for each $s \in children_{\mathcal{T}}(q)$. Thus, if $\mathcal{A}_{\pi_q}(s) \cap B_{len(q)}(s) \neq \emptyset$, for some $s \in children_{\mathcal{T}}(q)$, then the node s could violate some binding in $B_{len(q)}(s)$. To maintain such information during its execution, the algorithm in Figure 5 which we are about to describe performs a suitable coloring of the nodes. In particular, for each $q \in Q_{\mathcal{T}}$, we define the value $color(q)$ which can assume the following values:

WHITE: The color of a node q is WHITE provided that it has not been already visited by the algorithm. Thus, during the initialization phase, $color(q)$ is set to WHITE, for each $q \in Q_{\mathcal{T}}$.

GREEN/RED: Suppose that the visit of node q has been completed and that a safe topological ordering π_q of $\mathcal{T}_q$ has been constructed. Then $color(q)$ is set to GREEN, provided that π_q does not violate any binding imposed by its predecessor, i.e. provided that $\mathcal{A}_{\pi_q} \cap B_{len(q)-1}(q) = \emptyset$, otherwise is set to RED.

The algorithm which constructs a trie $\mathcal{T}'$ equivalent to a given input trie $\mathcal{T}$ and such that $\mathcal{T}'$ is endowed with a weakly safe topological ordering is shown in Figure 5. It performs a DFS visit of the trie $\mathcal{T}$, starting from $root(\mathcal{T})$. When the visit of a node $q \in Q_{\mathcal{T}} \setminus \{root(\mathcal{T})\}$ has been completed, a safe topological ordering π_q for the current subtrie rooted in q has been computed. The procedure for visiting a node $q \in Q_{\mathcal{T}}$ works in the following 6 main steps:

STEP 0 (Initialization)
During initialization, $\mathcal{A}(q)$ is set to $\emptyset$ and the ordering π_q is indirectly initialized by putting $\phi_q(0) = q$ (we recall that $\phi_q = \pi_q^{-1}$).

```
CONSTRUCT-SAFE-TOPOLOGICAL-ORDERING (T)
 1.  for each q ∈ Q_T do color(q) =WHITE
 2.  φ(0) = root(T)
 3.  i = 1
 4.  for each q ∈ children_T(root(T)) | color(q) = WHITE do
 5.      φ_q = VISIT(q, T)
 6.      for j = 0 to |Q_{T_q}| − 1 do φ(i + j) = φ_q(j)
 7.      i = i + |Q_{T_q}|
 8.  return (φ, T)

VISIT (q, T)
  /* STEP 0 (Initialization) */
 1.  φ_q(0) = q, i = 1
 2.  A(q) = ∅
  /* STEP 1 (Recursive calls) */
 3.  for each s ∈ children_T(q) | color(s) = WHITE do VISIT(s, T)
 4.  Green(q) = {s ∈ children_T(q) | color(s) = GREEN }
 5.  Red(q) = {s ∈ children_T(q) | color(s) = RED }
  /* STEP 2 (Resolving nodes of set Green(q)) */
 6.  if Green(q) ≠ ∅ then
 7.      Let s ∈ Green(q) | bind(lbl(q, s)) ≥ bind(lbl(q, p)), ∀ p ∈ Green(q)
 8.      for j = 0 to |Q_{T_s}| − 1 do φ_q(i + j) = φ_s(j)
 9.      i = i + |Q_{T_s}|
10.      A(q) = A(q) ∪ A(s)
11.      Green(q) = Green(q) − {s}
12.  for each s ∈ Green(q) do
13.      for j = 0 to |Q_{T_s}| − 1 do φ_q(i + j) = φ_s(j)
14.      i = i + |Q_{T_s}|
15.      A(q) = A(q) ∪ A(s) ∪ {lbl(q, s)}
  /* STEP 3 (Resolving nodes of set Red(q)) */
16.  for each s ∈ Red(q) do
17.      if lbl(q, s) ∉ A(q) then
18.          Red(q) = Red(q) − {s}
19.          for j = 0 to |Q_{T_s}| − 1 do φ_q(i + j) = φ_s(j)
20.          i = i + |Q_{T_s}|
21.          if A(q) = ∅ then A(q) = A(q) ∪ A(s)
22.          else A(q) = A(q) ∪ A(s) ∪ {lbl(q, s)}
  /* STEP 4 (Pruning all remaining red nodes) */
23.  for each s ∈ Red(q) do
24.      construct a new trie T' for lbl(s)
25.      for each u ∈ Q_{T'} do color(u) =WHITE
26.      prune T_s from T and insert it at the end of T'
27.      merge root(T') with root(T)
  /* STEP 5 (Setting color of node q) */
28.  if A(q) ∩ B_{len(q)−1}(q) = ∅ then color(q) = GREEN
29.  else color(q) = RED
30.  return φ_q
```

Figure 5: The algorithm for computing a safe topological ordering of the trie $\mathcal{T}$.

STEP 1. (Recursive calls)
After initialization, all $s \in children_{\mathcal{T}}(q)$ which have not been already visited

are visited. Then, at the end of Step 1, it follows inductively that, for each $s \in children_{\mathcal{T}}(q)$, a safe topological ordering π_s has been defined and either $color(s) =$ GREEN or $color(s) =$ RED.

STEP 2. (Resolving nodes of set $Green(q)$)
Suppose $Green(q)$ and $Red(q)$ are the sets of, respectively, GREEN and RED nodes of $children_{\mathcal{T}}(q)$. By construction, no node in $Green(q)$ violates any binding imposed by q. Thus, it is more convenient to resolve first the nodes in $Green(q)$ and later the ones in $Red(q)$. If $Green(q) \neq \emptyset$, a node $s \in Green(q)$ such that $lbl(q, s)$ has the largest binding value $bind(lbl(q, s))$ is selected. In this way all engaged edges which could violate the binding closest to q are eliminated. Then the topological ordering π_s is concatenated to π_q, the edge (q, s) becomes a 1-bit edge in π_q, and $\mathcal{A}(q)$ is set to the value $\mathcal{A}(q) \cup \mathcal{A}(s)$.
For each remaining node $s \in Green(q)$, the ordering π_s is concatenated to π_q, so that all engaged nodes in π_s become engaged nodes in π_q. Observe that, after the first selection, the edge (q, s) is a *long-bit edge* of π_q, so that $\mathcal{A}(q)$ must be set to the value $\mathcal{A}(q) \cup \mathcal{A}(s) \cup \{lbl(q, s)\}$.

STEP 3. (Resolving nodes of set $Red(q)$)
After that all GREEN nodes have been resolved in Step 2, nodes in $Red(q)$ are also resolved. In particular, if $Red(q) \neq \emptyset$, then an attempt is made to select a node $s \in Red(q)$ such that the symbol $lbl(q, s)$ is not engaged in π_q, i.e. $lbl(q, s) \notin \mathcal{A}(q)$. If such a node s is found, the ordering π_s is concatenated to the ordering π_q and the set $\mathcal{A}(q)$ of engaged nodes in π_q is updated accordingly. Step 3 is reapeted until no further node $s \in Red(q)$ can be selected.
Observe that, if $Green(q) = \emptyset$ at the beginning of Step 2, then the first selected node in $Red(q)$ generates a 1-bit edge in π_q. This case is tested in lines 21-22.

STEP 4. (Pruning all remaining RED nodes)
If $Red(q) \neq \emptyset$ after Step 4, each subtree rooted at any node $s \in Red(q)$ is first detached from $\mathcal{T}$ and then re-attached to $\mathcal{T}$ through a freshly introduced linear path labeled by $lbl(s)$. Notice that Step 4 can cause the trie $\mathcal{T}$ to become nondeterminstic.

STEP 5. (Setting color of node q)
Finally, if the engaged symbols of q violate some binding in $B(q)_{len(q)-1}$, i.e. $\mathcal{A}(q) \cap B(q)_{len(q)-1} \neq \emptyset$, $color(q)$ is set to RED. Otherwise $color(q)$ is set to GREEN.

At the end of the execution, the modified $\mathcal{T}$ and the function ϕ are returned. It turns out that ϕ^{-1} is a weakly safe topological ordering of $\mathcal{T}$.

Observe that there exist sets of patterns whose minimal tries admit no weakly safe topological ordering. The pruning of sub-tries in Step 4 is just intended to separate in $\mathcal{T}$ those patterns which cause troubles.

Let $\mathcal{P}$ be a set of patterns and let $\mathcal{T}$ be the minimal trie for $\mathcal{P}$. We evaluate the complexity of the algorithm in Figure 5 in terms of $L = size(\mathcal{P})$.

An efficient implementation of the algorithm CONSTRUCT-SAFE-TOPOLOGICAL-ORDERING maintains, for each node $q \in Q_{\mathcal{T}}$, the sets $B(q)_{len(q)-1}$ and $\mathcal{A}(q)$ in two

bit-vectors. Thus, if we assume that $|children_{\mathcal{T}}(q)| \leq \omega$, for each $q \in Q_{\mathcal{T}}$, where ω is the length of a computer word, the operations of set union and set intersection can be performed in constant time and $\mathcal{O}(|Q_{\mathcal{T}}|)$ space. Such assumption is quite reasonable, since in practical cases the degree of a node is rarely greater than ω. This is especially true if the patterns belong to a natural language where consecutive symbols are not independent, rather they are strongly related in most cases. For instance the symbol "q" is almost always followed by the symbol "u", whereas in general the symbol "t" is followed only by the symbols "a,e,h,i,l,o,r,u,y".

Additionally, if we maintain the topological orderings π_q, for each node q, as linked-lists, the operations in lines 8, 13, and 19, which concatenate two different topological orderings, can be also performed in constant time.

The procedure VISIT is called only once for each node $q \in Q_{\mathcal{T}}$. Since each node $s \in Q_{\mathcal{T}}$, with the exception of the root, will enter either set $Green(q)$ or set $Red(q)$, for only one node $q \in Q_{\mathcal{T}}$, we have that

$$\sum_{q \in Q_{\mathcal{T}}} (|Green(q)| + |Red(q)|) = |Q_{\mathcal{T}}| - 1\,.$$

Thus the overall complexity of Steps 2 and 3 is $\mathcal{O}(L)$, since $|Q_{\mathcal{T}}| = \mathcal{O}(L)$.

In Step 4, the pruning of a RED node s consists in following the path from the root of the trie to node s. Thus the overall work of Step 4 is bounded again by $\mathcal{O}(L)$.

Finally Step 0 and Step 5 are performed in constant time. Thus, it turns out that the algorithm CONSTRUCT-SAFE-TOPOLOGICAL-ORDERING has a $\mathcal{O}(L)$-time and -space complexity.

It must be remarked that in general the algorithm CONSTRUCT-SAFE-TOPOLOGICAL-ORDERING does not construct the *minimal* trie $\mathcal{T}'$, equivalent to a given trie $\mathcal{T}$, which is endowed with a weakly safe topological sorting. A natural variant which enforces minimality takes quadratic time.

On the other hand, some experimentations has shown that the heuristics embodied in Steps 2, 3, and 4 are quite effective in keeping the returned trie close to minimal.

6 Conclusion

In this paper we have presented a new algorithm for the multiple pattern matching problem, based on the bit-parallelism technique. In particular, our algorithm is based on the parallel simulation of a factor-based trie (not necessarily the optimal one) for the input set of patterns. In fact, our simulation requires that the factor-based trie admits a topological ordering which is weakly safe, in a sense amply explained before. The complexity of our algorithm is linear in the length of the text and in the size of the set of patterns.

We have also shown how to transform a given minimal trie into a trie which has a weakly safe topological ordering in linear time and space in the size of the set of patterns. The resulting trie is in general significantly smaller than the maximal tries used in the other multi-pattern matching algorithms based on bit-parallelism.

Further variations and improvements are still possible. For instance, we expect that our approach can be extended to obtain a space efficient version of the BNDM algorithm for the multiple pattern matching problem.

An interesting open problem is to find other suitable topological orderings on deterministic tries which guarantee that they can be easily simulated by bit-parallelism, without any need to modify their topology.

References

[AC75] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975.

[BM77] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.

[BYG92] R. A. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Commun. ACM*, 35(10):74–82, 1992.

[CCG+93] M. Crochemore, A. Czumaj, L. Gąsieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter. Fast practical multi-pattern matching. Rapport 93-3, Institut Gaspard Monge, Université de Marne la Vallée, 1993.

[CCG+94] M. Crochemore, A. Czumaj, L. Gąsieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter. Speeding up two string matching algorithms. *Algorithmica*, 12(4/5):247–267, 1994.

[CR94] M. Crochemore and W. Rytter. *Text algorithms*. Oxford University Press, 1994.

[CW79] B. Commentz-Walter. A string matching algorithm fast on the average. In H. A. Maurer, editor, *Proceedings of the 6th International Colloquium on Automata, Languages and Programming*, number 71 in Lecture Notes in Computer Science, pages 118–132, Graz, Austria, 1979. Springer-Verlag, Berlin.

[Hor80] R. N. Horspool. Practical fast searching in strings. *Softw. Pract. Exp.*, 10(6):501–506, 1980.

[KMP77] D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(1):323–350, 1977.

[NR98] G. Navarro and M. Raffinot. A bit-parallel approach to suffix automata: Fast extended string matching. In M. Farach-Colton, editor, *Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching*, number 1448 in Lecture Notes in Computer Science, pages 14–33, Piscataway, NJ, 1998. Springer-Verlag, Berlin.

[Raf97] M. Raffinot. On the multi backward dawg matching algorithm (MultiBDM). In R. Baeza-Yates, editor, *Proceedings of the 4th South American Workshop on String Processing*, pages 149–165, Valparaiso, Chile, 1997. Carleton University Press.

[WM91] S. Wu and U. Manber. Fast text searching with errors. Report TR-91-11, Department of Computer Science, University of Arizona, Tucson, AZ, 1991.

Compressed Pattern Matching in JPEG Images

Shmuel T. Klein[1] and Dana Shapira[2]

[1] Dept. of Computer Science
Bar Ilan University
Ramat-Gan 52900, Israel
e-mail: tomi@cs.biu.ac.il

[2] Dept. of Computer Science
Ashkelon Academic College
Ashkelon, Israel
e-mail: shapird@acad.ash-college.ac.il

Keywords: Data Compression, JPEG, Huffman

Abstract. The possibility of applying compressed matching in JPEG encoded images is investigated and the problems raised by the scheme are discussed. A part of the problems can be solved by the use of some auxiliary data which yields various time/space tradeoffs. Finally, approaches to deal with extensions such as allowing scaling or rotations are suggested.

1 Introduction

The paradigm of *compressed pattern matching* has recently gotten a lot of attention. The idea of the compressed matching was first introduced in the work of Amir and Benson [1] as the task of performing pattern matching in a compressed text without decompressing it. For a given text T and pattern P and complementary encoding and decoding functions, $\mathcal{E}$ and $\mathcal{D}$ respectively, our aim is to search for $\mathcal{E}(P)$ in $\mathcal{E}(T)$, rather than the usual approach which searches for the pattern P in the decompressed text $\mathcal{D}(\mathcal{E}(T))$. Amir and Benson deal with a run-length encoded two-dimensional pattern, but most works address the problem of finding one-dimensional patterns in files compressed by various methods, such as Huffman coding [9], Lempel-Ziv [13], or specially adapted methods [11, 8].

We concentrate here on two-dimensional compressed matching in which the given text is an image encoded by the standard JPEG baseline scheme [6] and the pattern consists of a given image fragment we are looking for. In a more general setting, a collection of images could be given, and the subset of those including at least one copy of the pattern is sought. An example for the former could be an aerial photograph of a city in which a certain building is to be located, an example for the more general case could be a set of pictures of faces of potential suspects, which have to be matched against some known identifying feature like a nose or an eyebrow.

Baseline JPEG uses a static Huffman code, without which compressed matching would not always be possible, since our underlying assumption is that all occurrences

of the pattern are encoded by the same binary sequence. This is not the case for dynamic Huffman coding or for arithmetic coding. Lempel-Ziv methods are also adaptive, but for them compressed matching is possible because all the fragments of the pattern appear in the text, though not necessarily in the same order as in the pattern.

In a first approach, we accept as simplifying assumption that only exact copies of the pattern are to be found. Returning to the example of the aerial picture, it would of course also be interesting to locate the given building if the pattern presents it in a different angle than it appears in the larger image, or at another scale, or even only partially, because it could have been occluded by a cloud when the picture has been taken. The corresponding pattern matching problems, allowing rotations, scaling and occlusions, are more difficult and have been treated in [2, 3].

In the next section, we review the basic ingredients of the JPEG algorithm, then turn in Section 3 to the method we suggest for compressed matching in JPEG files. The main problem to be dealt with is one of synchronization and alignment, so we explore in Section 4 the possibility of using auxiliary files to solve such alignment problems. The last section deals with extensions to rotations and scaling.

2 The JPEG standard

JPEG [6] is a lossy image compression method. In a first step, the picture is split into a sequence of blocks of size 8×8 pixels. Each block is then compressed by the following sequence of transformations:

1. Applying a *Discrete Cosine Transform* (DCT) [14] to the set of 64 values of the pixels in the block;

2. Applying *Quantization* to the DCT coefficients, thereby producing a set of 64 smaller integers. This step causes a loss of information but makes the data more compressible;

3. Applying an *entropy encoder* to the quantized DCT coefficients. Baseline JPEG uses Huffman coding in this step, but the JPEG standard specifies also arithmetic coding as possible alternative.

The decompression process just reverses the actions and their order. It first applies Huffman decoding, then dequantizes the coefficients, and finally uses an inverse DCT to obtain a set of values. Because of the quantization step, the reconstructed set includes only approximated values.

The coefficient in position (0,0) (left upper corner) is called the DC coefficient and the 63 remaining values are called the AC coefficients. In principle, the DC coefficient should store a measure of the average of the 64 pixel values of the given block, but since there is usually a strong correlation between the DC coefficients of adjacent blocks, what is actually stored is the difference between the average in this block and the average in the previous one.

Baseline JPEG uses two different Huffman trees to encode the data. The first encodes the lengths in bits (1 to 11) of the binary representations of the values in the DC fields. The second tree encodes information about the sequence of AC coefficients.

As many of them are zero, and most of the non-zero values are often concentrated in the upper left part of the 8×8 block, the AC coefficients are scanned in a fixed zig-zag order, processing elements on a diagonal close to the upper left corner before those on such diagonals further away from that corner; that is, the order is given by (0,1), (1,0), (2,0), (1,1), (0,2), (0,3), (1,2), etc. The second Huffman tree encodes pairs of the form (n, ℓ), where n (limited to the range 0 to 15) is the number of elements that are 0, preceding a non-zero element in the given order, and ℓ is the length in bits (1 to 10) of the binary representation of the non-zero quantized AC value. The second tree includes also codewords for End of Block (EOB), which is used when no non-zero elements are left in the scanning order, and for a sequence of 16 consecutive 0s in the AC sequence (ZRL). The Huffman trees used in baseline JPEG are static, and can be found in [15].

Each 8×8 block is then encoded by an alternating sequence of Huffman codewords and binary integers (except that the codewords for EOB and ZRL are not followed by any integer), the first codeword belonging to the first tree and relating to the DC value, the other codewords encoding the (n, ℓ) pairs for the AC values, with the last codeword in each block representing EOB. Figure 1(a) brings an example block of quantized values, with the DC value in boldface in the upper left corner. The upper part of Figure 1(b) shows the encoding of this block, with elements to be Huffman encoded appearing in parentheses, and the elements corresponding to DC (the value of which we assume to be 5) bold faced; the binary translation of the encoding, with framed Huffman codewords, is shown underneath.

20	1	0	0	0	0	0	0
0	3	0	0	0	0	0	0
0	0	0	0	0	0	0	0
-2	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

(a) Typical JPEG block

(3) **5** (0,1) 1 (2,2) 3 (4,2) -2 (EOB)

100 101 00 1 11111001 11 1111111000 01 1010

(b) Encoding of JPEG block

FIGURE 1: *Example of JPEG block and its encoding*

JPEG encodes the blocks row by row, from left to right, and concatenates the encoded blocks. A small header encodes the number of rows and columns, so there is no need to encode an end-of-row indicator specifically. Actually, to simplify the discussion and the examples, our description refers to only one component, the *luminance*, of JPEG encoding, which corresponds to black and white images. JPEG also supports color images, where each color pixel is split into several components (RGB or YUV).

3 Pure compressed matching

We are given an image T of $n \times k$ pixels, in which a two-dimensional pattern P of size $m \times \ell$ pixels should be found. Since JPEG works with 8×8 pixel blocks, we

assume that n and k are multiples of 8. The compressed matching starts by encoding the pattern using the same JPEG algorithm as the one used for the original image. Even then we cannot assure that a pattern can be located, as the 8×8 blocks of the pattern are not necessarily aligned with those of the image. The search process has therefore to be repeated 64 times, positioning, for each matching attempt, the leftmost uppermost pixel of the first 8×8 block in the pattern at the ith pixel in the jth row, $1 \leq i, j \leq 8$. Figure 2 is an example of how the pattern could be partitioned: there will usually be a frame at the border of the pattern (the darker area in Figure 2) corresponding to 8×8 blocks that fit only partially. The pixels in this frame will not participate in the matching process, so the pattern is actually restricted to an area of full contiguous 8×8 blocks (the white area in Figure 2). For the rest of this paper, let m and ℓ then represent the dimensions of the restricted pattern, that is, m and ℓ are also multiples of 8.

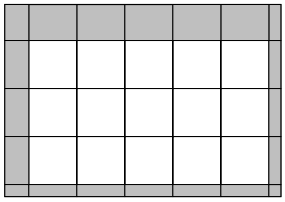


FIGURE 2: *Example of partition of the pattern into* 8×8 *blocks*

The first block is JPEG encoded, yielding one DC value and a sequence of AC values. Since DC elements are encoded relative to preceding blocks, the DC value of this first block cannot be located, so the matching starts only from the beginning of the sequence of AC values. These are calculated for each block independently, therefore if the pattern-block does appear as a block in the image T, the encoded sequence of AC values will appear in the encoded image $\mathcal{E}(T)$. The DC values of the second and subsequent blocks in the first row of 8×8 blocks of P can be evaluated based on the DC values of the preceding blocks, hence the first part of the encoded pattern to be searched for in $\mathcal{E}(T)$ consists of the sequence of AC values followed by the $\ell/8 - 1$ encoded 8×8 blocks of the first row.

The compressed matching paradigm raises then several problems. First, suppose that the binary sequence encoding the first part of the pattern is indeed located. This does not necessarily mean that an occurrence of the encoded elements is found, as the beginning of the Huffman codewords might not be synchronized. Consider, for example, the block **(2) 3** (0,2) −3 (EOB), to be located in a sequence of several blocks identical to those of the example in Figure 1(b). Figure 3 shows that the pattern (after having stripped the DC values) will be found erroneously crossing the block boundaries in $\mathcal{E}(T)$.

The same problem was noted in [9], and in [10] in an application to parallel decoding of a JPEG file when several processors are available. For long enough patterns, the tendency of Huffman codes to resynchronize after errors may suggest that false alarms as those in the above example might be rare, but in our application, the rows of the pattern may be short. Moreover, the problem in the JPEG case is more severe than for plain Huffman decoding. For the latter, once synchronization has been regained, the remainder of the encoded file is correct. In JPEG files, on

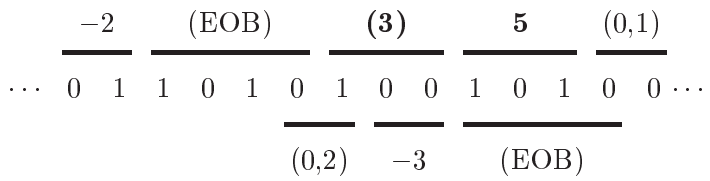


FIGURE 3: *Example of false alignment*

the other hand, consisting of both Huffman codewords and integer encodings, the fact that a given bit is the last in a codeword for both the correct and the erroneous decoding does not imply that both decodings will continue identically. Referring again to Figure 3, the codewords for **(3)** and -3 end at the same bit, which is nevertheless not a synchronization point.

The second problem is that the encoded pattern does not appear consecutively in the encoded image (unless $k = \ell$), but with gaps corresponding to the encoding of $(k - \ell)/8$ blocks. The pattern is therefore partitioned into $m/8$ sub-patterns, each corresponding to a row of $\ell/8$ blocks, and with the first DC value of each sub-pattern eliminated. If the sub-patterns are located using some pattern matching algorithm, we cannot conclude with certainty that the pattern has been found. In addition to the above problem of possible false alignments, one cannot know if each of the gaps are indeed the encoding of $(k - \ell)/8$ blocks, even if the sub-patterns are found in the required order and even if all of them are true matches.

One of the possible solutions could be, once the first row of the pattern has been found, to continue decompressing the image, keeping a count of the decoded blocks. In other words, pattern matching would only be used for the first row of the pattern, then the image would be decoded sequentially. In fact, one does not really need full decoding: the Huffman codewords in the JPEG file indicate the length in bits of the integers following the codewords, and for our purpose, these integers can be simply skipped. This solution could, however, not really be considered as compressed matching, since, depending on the position of the first occurrence of the pattern in the encoded file, large parts of it, possibly almost the whole original file, are decompressed.

The third problem relates to the fact that there are possibly many occurrences of the pattern, perhaps even overlapping ones. In images this might be more frequent than for plain texts, because large areas could represent some uniform background (a blue sky, dark parts in the shadow, etc.), and therefore consist of many identical blocks. If each of the rows of the pattern is located several times, we need to match somehow their occurrences to check whether indeed we have an occurrence of the whole pattern. This might be a difficult task even if we ignore the problem of certain occurrences being false matches.

We therefore conclude that compressed pattern matching in JPEG files is hard to achieve, unless we keep some auxiliary data, as suggested in the following section.

4 Compressed matching with auxiliary data

The task would be much easier if one would know, for a given position in the JPEG encoded file, the index of the corresponding 8×8 block in the original file. A step in this direction would be using synchronizing codewords (see [7]), for example at the

end of each encoded row, but this would require a change in the encoding standard, for example to JPEG-2000 [12] which has synchronizing codewords built-in. In fact, the code used in baseline JPEG is not really a Huffman code, because it is not complete: there is, e.g., no codeword consisting only of 1's. This can be exploited to devise a *synchronizing sequence*: the longest sequence of 1's that can appear is of length 29, in the encoding of (10,10) 1023 (15,10), which is translated into 1111111111001111 1111111111 1111111111111110. Therefore a sequence of 30 consecutive 1's is synchronizing. This synchronizing sequence could be inserted at the end of each row, which could therefore be detected without decoding. Alternatively, instead of wasting 30 bits for synchronization, one of the codewords could be replaced by this string of 1's, for example the codeword 1010 for EOB. This would increase each encoded block by 26 bits, but false matches are then easily detected. Nevertheless, 26 bits for each 8×8 block, which are generally encoded by a few hundred bits or less, might be too high a price to pay.

4.1 Building an index

Instead of modifying the JPEG file, one could construct a table S, acting as an index, that would be stored in addition to the original compressed file. $S(i)$ would be the bit-position, within the JPEG image, of the beginning of the encoding of the AC sequence in the ith block, that is the index of the bit following the DC value. The size of each entry in S would be $\lceil \log_2 |\mathcal{E}(T)| \rceil$, where $|x|$ refers to the size of x in bits, so that a 3 byte entry could accommodate a compressed image of size up to 2MB. The number of entries is S is $nk/64$, the number of 8×8 blocks in T.

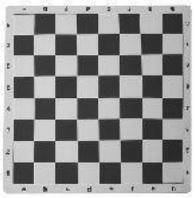
The construction of such an index has to be done in a preprocessing stage, and it could be argued that this contradicts the main idea of compressed matching, since while building the table S one actually decompresses the whole image. Nevertheless, the preprocessing can be justified in certain applications, for example when one large image will be used many times for searches with different patterns. This is similar to regular pattern matching with a fixed large text of size n and possibly many patterns to be looked for. Some of the fastest algorithms are then based on constructing a *suffix tree* [16, 4], the size of which may often exceed that of the text itself. Construction time is linear in n, but once the suffix tree is ready, the time to locate a pattern is independent of the size of the text.

The index S can be used to solve some of the problems mentioned above. Once the encoding of the first row of the pattern image has been located in $\mathcal{E}(T)$ at bit offset y, a binary search for y in S can decide in $\lceil \log n + \log k \rceil - 6$ comparisons whether the match is a true one. Similar searches for the following rows of the pattern can locate all the rows, without decoding.

To get a feeling about the size of the required indices, we have applied this idea on the three grayscale sample JPEG files in Figure 4: the classical Lenna picture, a chessboard with many identical sub-parts, and a rose. Table 1 shows the details, giving the number of rows and columns, $r \times c$, the size in bytes, s, of the compressed file, and the absolute (in bytes) and relative size (in percent) of the index S. The size is given by $((\lceil r/8 \rceil \cdot \lceil c/8 \rceil)(\log_2(s) + 3))/8$.

If the size of S is too large, a time/space tradeoff can be obtained by fixing an integer parameter d and storing only every dth entry of S. The storage overhead is

Lenna



Chess

Rose

FIGURE 4: *Examples of JPEG files*

File	pixels	jpeg size	index size	%
Lenna	256 × 256	30,763	2304	7.5
Chess board	150 × 150	14,112	768	5.4
Rose	227 × 149	12,089	1171	9.7

TABLE 1: *Details on sample files*

reduced by a factor of d, at the cost of increased search time: the binary search for the bit offset y now locates the largest value is S that is still smaller or equal to y; from there, up to d blocks have to be decoded. For example, the index for the Lenna picture can be reduced to less than 1% if only every eighth block is indexed, and if one records only the beginning of every row, the index reduces to 72 bytes.

4.2 Dealing with multiple matches

We now turn to the possibility of having found many matches for each of the rows of the pattern. Using the table S, each of the found offsets is checked to correspond to a true match and then translated to a block index. Since the dimensions of the image T are known, each index can be translated into an (r, c) pair, denoting the indices of the corresponding row and column. Let (R_i, C_i) be the sequence of n_i (true) occurrences of the ith row of the pattern,

$$(R_i, C_i) = \{(r_{i1}, c_{i1}), (r_{i2}, c_{i2}), \ldots, (r_{in_i}, c_{in_i})\}, \quad 1 \leq i \leq m.$$

The sequences can be kept in lexicographically increasing order. We need to check whether consecutive rows of the pattern have appearances in consecutive rows and identical columns of the image. Formally, we seek

$$\bigcap_{i=1}^{m} (R_i - i + 1, C_i),$$

where we use the notation $A - x$ for a set of integers $A = \{a_1, \ldots, a_n\}$ and an integer x to stand for the set $\{a_1 - x, \ldots, a_n - x\}$.

The following algorithm uses m pointers, one for each of the sequences, to find all the occurrences:

Repeat until one of the sequences is exhausted
    find the smallest element (r, c) in $(R_1, C_1) \cap (R_2 - 1, C_2)$ by sequential search
    for $i \leftarrow 3$ to m
        search for an occurrence of (r, c) in $(R_i - i + 1, C_i)$
    if (r, c) is common to all m sequences, increase all m pointers by 1

The search in the iterative step can be done by binary search, since the sequences are ordered, but this is not necessarily the best solution. Consider the special case in which all n_i are equal to n_1, and h elements are found in the intersection $(R_1, C_1) \cap (R_2 - 1, C_2)$. Assume also that $h > n_1/\log n_1$ and that all h elements of the intersection belong to the first halves of both (R_1, C_1) and $(R_2 - 1, C_2)$. Then performing the intersection takes $2n_1$ comparisons, and each of the h searches in each of the $m-2$ remaining sequences requires $\log n_1$ comparisons. To reduce this number even by 1, the length of the sequence has to be cut at least to half, so even reducing the search to the remaining sequence after each located element wouldn't help in our case. The total search time would thus be $2n_1 + h\,(m-2)\log n_1 > n_1 m$. On the other hand, scanning the m lists sequentially can be done in time $n_1 m$.

Note that it would pay to start the process by intersecting the two shortest lists, rather than the two first, which would tend to reduce h. Moreover, the intersection could be done by binary merge [5] rather than linearly.

In an experiment run on each of the images of Figure 4, a random 15 byte long fragment of the encoded file was taken as pattern, corresponding to a part of a row of the image, and occurrences of this pattern were sought. In each case, only a single occurrence was found, corresponding to the true match. This suggests that in many real life JPEG files, multiple matches will not cause a problem. On the other hand, we repeated the test with a pure black bitmap file, and found there many matches, as expected.

5 Matching with scaling and rotations

Consider the problem of locating the pattern P after having scaled it by a factor α and/or rotated it by an angle γ. The one to one correspondence between 8×8 blocks of pattern and image might be lost, but since the DCT transforms the full block as one indivisible entity, there is no way to detect the encoding of parts of the block in the JPEG file. So instead of trying to transform the encoded pattern, one has to transform the pattern first, and then apply the encoding.

For $\alpha < 1$, both height and width of the occurrence of pattern P in the image T should be α times smaller than in P. Since it is the pattern that is encoded, we get the requested effect by *enlarging* the pattern by a factor of $\beta = 1/\alpha$ before applying JPEG. If β is an integer, one could duplicate each pixel in each row, as well as the such enlarged rows β times. The resulting pattern is of lower quality than a possible occurrence in the given image, so some smoothing, taking neighboring pixels into account, could improve the search, but the DCT will take care, at least partially, of the smoothing anyway. If β is not an integer, certain rational factors can be obtained by a process similar to the one depicted in Figure 5(a). For $\beta = 1.5$, transform each 2×2 block into a 3×3 block, inserting the missing values (in grey) by interpolation.

If $\alpha > 1$, the pattern has to be reduced by a factor of $\beta = 1/\alpha$. If α is an integer, the simplest way to proceed is taking every αth pixel in both dimensions. A more precise way would be to consider some or all translations of such subsets of the pattern having their pixels α positions apart, and averaging among them the value for each pixel. For certain non-integer values of α, one could proceed similarly to the above non-integer case for β.

As to rotations, if γ is a multiple of a right angle, say 90°, 180° or 270°, each

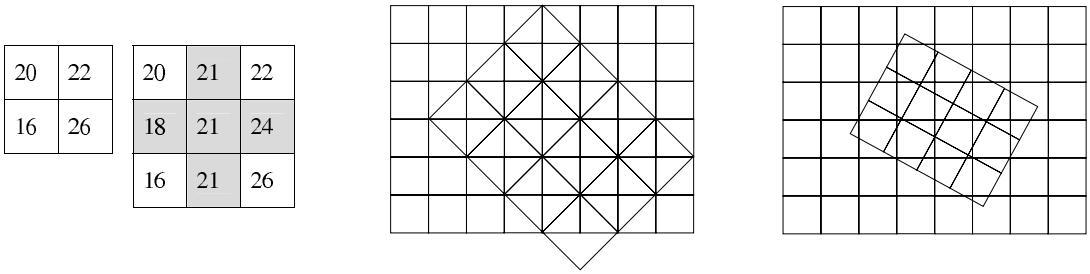


FIGURE 5: *Examples of possible rotations*

8×8 matrix can be transposed or reversed accordingly, thereby redefining the rows and columns of the pattern. If $\gamma = 45°$ after a scaling of $\alpha = \sqrt{2}$, as in Figure 5(b), each pattern block would have to match four halves of image blocks, but even if there is no such regularity and the pattern blocks might intersect a varying number of image blocks in various layouts, as for example in Figure 5(c), one can deal with it by rotating first the pattern by $-\gamma$, then partitioning into blocks and encoding.

Conclusion

Searching directly in JPEG encoded images seems to be a difficult task because the blocking used, as well as the DCT applied to the blocks, does not allow any interaction between adjacent blocks. Using an index, the size of which can be controlled in a time/space tradeoff, may alleviate some of the problems.

References

[1] AMIR A., BENSON G., Efficient two-dimensional compressed matching, *Proc. Data Compression Conference DCC–92*, Snowbird, Utah (1992) 279–288.

[2] AMIR A., BUTMAN A., CROCHEMORE M., LANDAU G.M., SCHAPS M., Two dimensional pattern matching with rotations, *Theoretical Computer Science*, **314**(1–2) (2004) 173–187.

[3] AMIR A., BUTMAN A., LEWENSTEIN M., PORAT E., Real two dimensional scaled matching, *Proc. WADS* (2003) 353–364.

[4] APOSTOLICO A., The myriad virtues of subword trees, *Combinatorial Algorithms on Words,* NATO ASI Series Vol **F12**, Springer Verlag, Berlin (1985) 85–96.

[5] HWANG F.K., LIN S., A simple algorithm for merging two disjoint linearly-ordered sets, *SIAM Journal of Computing* **1** (1972) 31–39.

[6] ISO/IEC 10918-1 Information Technology - Digital Compression and Coding of Continuous–Tone Still Images Requirements and Guidelines, *International Standard ISO/IEC*, Geneva, Switzerland (1993).

[7] Ferguson T.J., Rabinowitz J.H., Self-synchronizing Huffman codes, *IEEE Trans. on Inf. Th.* **IT–30** (1984) 687–693.

[8] Klein S.T., Shapira D., A new compression method for compressed matching, *Proc. Data Compression Conference DCC–2000*, Snowbird, Utah (2000) 400–409.

[9] Klein S.T., Shapira D., Pattern Matching in Huffman Encoded Texts, *Information Processing and Management* **41** (2005) 829–841.

[10] Klein S.T., Wiseman Y., Parallel Huffman Decoding with Applications to JPEG Files, *The Computer Journal* **46**(5) (2003) 487–497.

[11] Manber U., A Text Compression Scheme That allows Fast Searching Directly in the compressed File, *ACM Trans. on Inf. Sys.* **15** (1997) 124–136.

[12] Marcellin M.W., Gormish M.J., Bilgin A., Boliek M.P., An Overview of JPEG-2000, *Proc. Data Compression Conference DCC-2000*, Snowbird, Utah (2000) 523–541.

[13] Navarro G., Raffinot M., A general practical approach to pattern matching over Ziv-Lempel compressed text, *Proc. 10th Symp. on Combinatorial Pattern Matching,* Warwick, UK, July 22–24 1999, *LNCS* **1645**, Springer Verlag, Berlin(1999) 14–36.

[14] Rao K.R., Yip P., *Discrete Cosine Transform Algorithms, Advatages, Applications*, Academic Press Inc., London (1990).

[15] Wallace G.K., The JPEG Still Picture Compression Standard, *Communication of the ACM* **34** (1991) 30–44.

[16] Weiner P., Linear pattern matching algorithms, *Proc. 14th Annual IEEE Symposium on Switching and Automata Theory*, Washington, DC, (1973) (1–11).

Bounded Size Dictionary Compression: Relaxing the LRU Deletion Heuristic

Sergio De Agostino

Computer Science Department
Università "La Sapienza"
Via Salaria 113, 00135 Roma, Italy

e-mail: deagostino@di.uniroma1.it

Abstract. The unbounded version of the Lempel-Ziv dynamic dictionary compression method is P-complete. Therefore, it is unlikely to implement it with sublinear work space unless a deletion heuristic is applied to bound the dictionary. The well-known LRU strategy provides the best compression performance among the existent deletion heuristics. We show experimental results on the compression effectiveness of a relaxed version (RLRU) of the LRU heuristic. RLRU partitions the dictionary in p equivalence classes, so that all the elements in each class are considered to have the same "age" for the LRU strategy. Such heuristic turns out to be as good as LRU when p is greater or equal to 2. Moreover, RLRU is slightly easier to implement than LRU in addition to be more space efficient.

Keywords: Bounded dictionary compression, space complexity, LRU strategy

1 Introduction

The Lempel-Ziv dynamic dictionary (LZ2) compression algorithm learns substrings by reading the input string from left to right with an *incremental parsing* procedure [7]. The dictionary is empty, initially. The procedure adds a new substring to the dictionary as soon as a prefix of the still unparsed part of the string does not match a dictionary element and replaces the prefix with a pair comprising a pointer to the dictionary and the last uncompressed character. For example, the parsing of the string $abababaaaaaa$ is a, b, ab, aba, aa, aaa and the coding is $0a, 0b, 1b, 3a, 1a, 5a$ (the pointer value for the first element in the dictionary is 1 and 0 represents the empty string). We will see in the next section different LZ2 compression heuristics (NC, FC, ID, AP), which work with a dictionary containing initially the alphabet characters and produce a coding with no raw characters.

The main issue for implementation purposes is to bound the work space to produce the incremental parsing of the string to compress. Since the problem of computing such parsing is P-complete [2, 3], it is unlikely to have sublinear work space when LZ2 compression is implemented unless a deletion heuristic is applied to bound the dictionary. Several deletion heuristics have been designed and applied to the compression heuristics mentioned above (see the books of Storer [5, 6] and Bell, Cleary

and Witten [1]). A strategy that can achieve good compression ratio with small memory is the LRU deletion heuristic that discards the least recently used dictionary element to make space for the new substring. The least recently used strategy provides the best compression performance among the well-known heuristics (FREEZE, RESTART, SWAP, LRU). AP-LRU turns out to be the best compression heuristic when the dictionary is bounded.

When the size of the dictionary is $O(\log^k n)$ the LRU strategy is log-space hard for SC^k (Steve Cook's class), the class of problems solvable simultaneously in polynomial time and $O(\log^k n)$ space [4]. Since its sequential complexity is polynomial in time and $O(\log^k n \log\log n)$ in space, the problem belongs to SC^{k+1}. Moreover, in [4] a relaxed version (RLRU) was introduced which turned out to be the first (and only so far) natural SC^k-complete problem. RLRU partitions the dictionary in p equivalence classes, so that all the elements in each class are considered to have the same "age" for the LRU strategy.

While in [4] the RLRU heuristic was considered only for theoretical reasons concerning complexity theory, in this paper we want to look at its practical aspects. We show experimental results on its compression effectiveness for $2 \leq p \leq 6$, using the AP compression heuristic. RLRU turns out to be as good as LRU even when p is equal to 2. Since RLRU removes an arbitrary element from the equivalence class with the "older" elements, the two classes (when p is equal to 2) can be implemented with a couple of stacks, which makes RLRU slightly easier to implement than LRU in addition to be more space efficient. Surprisingly, the compression effectiveness (which we can measure as the inverse of the compression ratio) is not monotonicly increasing with the value of p. This might be explained by the fact that the approach is heuristic (choosing to remove an older element is not always a better choice). However, LRU is always strictly better (in an irrelevant way for the compression effectiveness) than RLRU. This fact shows that there should be always an improvement when two values of p differ substantially.

Simpler choices for the deletion heuristic are FREEZE, RESTART and SWAP. These heuristics do not delete elements from the dictionary at each step. SWAP is the best among these simpler approaches and has a worse compression performance than RLRU and LRU. We describe compression and deletion heuristics in section 2. In section 3, we discuss the complexity of the LRU and RLRU heuristics. In section 4, we compare the experimental results of LRU, RLRU and SWAP. Conclusions are given in section 5.

2 Compression and Deletion Heuristics

As mentioned in the introduction, the compression and deletion heuristics presented in this section can be found in [1, 5, 6]. The incremental parsing procedure used by the LZ2 algorithm produces a compressed string comprising pointers and raw characters. In practice, we do not want to leave characters uncompressed. This can be avoided by initializing the dictionary with the alphabet characters. The NC (next character) heuristic also parses the string from left to right with a greedy procedure. It finds the longest match in the current position and updates the dictionary by adding the concatenation of the match with the next character. The FC (first character) heuristic differs in the way it updates the dictionary. The element to add is defined

as the concatenation of the last match with the first character of the current match. With the ID (identity) heuristic, the element to add is defined as the concatenation of the last match with the whole current match. The AP (all prefixes) heuristic adds a set of elements to the dictionary at each step. Each element is the concatenation of the last match with a prefix of the current match. In this way, the dictionary of the AP heuristic has both the characteristics of the dictionaries of the FC and ID heuristics. Observe that with FC, ID and AP, an element to add might be in the dictionary already. How these heuristics work on the example in the introduction is shown in Figure 1.

NC heuristic

parsing: a, b, ab, aba, a, aa, aa;

dictionary: $a, b, ab, ba, aba, abaa, aa, aaa$;

coding: 1, 2, 3, 5, 1, 7, 7

FC heuristic

parsing: a, b, ab, ab, a, a, aa, aa

dictionary: a, b, ab, ba, aba, aa, aaa, aaaa

coding: 1, 2, 3, 3, 1, 1, 7, 7

ID heuristic

parsing: a, b, ab, ab, a, a, aa, aa

dictionary : a, b, ab, bab, abab, aba, aa, aaaa

coding: 1, 2, 3, 3, 1, 1, 7, 7

AP heuristic

parsing: a, b, ab, ab, a, a, aa, aa

dictionary : a, b, ab, ba, bab, aba, abab, aa, aaa, aaaa

coding: 1, 2, 3, 3, 1, 1, 8, 8

Figure 1: The compression heuristics.

It is well known that these heuristics can be implemented by storing the dictionary in a tree data structure, called *trie*. At each step, we find the longest match in the dictionary as a path from the root to a leaf of the trie and update the dictionary by adding a new leaf to the trie. Real time implementations are possible for each compression heuristic using any deletion heuristic (FREEZE, RESTART, SWAP, LRU and RLRU) to bound the dictionary. FREEZE, RESTART and SWAP work as it follows:

- FREEZE: once the dictionary is full, freeze it and do not allow any further entries to be added.

- RESTART: stop adding further entries when the dictionary is full; when the compression ratio starts deteriorating clear the dictionary and learn new strings.
- SWAP: when the *primary* dictionary first becomes full, start an *auxiliary* dictionary, but continue compression based on the primary dictionary; when the auxiliary dictionary becomes full, clear the primary dictionary and reverse their roles.

The SWAP and RESTART heuristics can be viewed as discrete versions of LRU. In fact, the dictionaries depend only on small segments of the input string.

parsing: $a, b, ab, ab, a, a, aa, aa$;

dictionary (step 3): a, b, ab, ba, bab

dictionary (step 4): a, b, ab, ba, aba

dictionary (step 4): a, b, ab, abab , aba

dictionary (step 6): a, b, ab, aa, aba

dictionary (step 7): a, b, ab, aa, aaa

dictionary (step 8): a, b, aaaa, aa, aaa

coding: 1, 2, 3, 3, 1, 1, 4, 4

Figure 2: The AP-LRU heuristic on the example string.

We showed in the introduction of the paper how the LZ2 algorithm parses the example string *abababaaaaaa*. If we bound the dictionary size with 3 and use LRU, after three steps a, b, ab is the partial parsing, $0a, 0b, 1b$ is the partial coding and the dictionary is filled up with the three elements a, b, ab. The LRU heuristic works as follows:

> LRU: define a string as "used" when it is added to the dictionary and remove the least recently used leaf of the trie representing the dictionary after a new leaf is added. The pointer to the element which is removed becomes the pointer to the new element.

Hence at the fourth step, first *aba* is added and coded as $3a$. Then, b is discarded. Finally, *aba* is replaced with *aa*, coded as $1a$, and *ab* with *aaa*, coded as 2a.

Observe that while for the NC heuristic the element added to the dictionary is an extension of the current match as for the original LZ2 algorithm, this is not true for the other heuristics. To make things work properly when we apply the LRU deletion strategy to the FC, ID and AP heuristics, a string is defined to be "used" also when it is matched. AP-LRU turns out to be the best compression heuristic when the dictionary is bounded. How the AP-LRU heuristic works on the example string with a dictionary of size 5 is shown in Figure 2. Steps correspond to the parsing. In this example, the AP-LRU heuristic adds more than one element only at the fourth parsing step. In Figure 3, we extend the example by adding the suffix *bbaaa* to make some observations. At step 11, the current match is removed from the dictionary. In

this case, the AP-LRU heuristic puts it back into the dictionary at step 12 and then it adds its extensions (this can happen with FC and ID as well). With AP, it could be possible that prefixes of the current match are removed and similarly they would be put back into the dictionary at the next step. Finally, observe that at step 10 if *aab* were removed instead of *aaa*, *aaa* would be parsed off at the end providing a shorter code for the string. This shows that removing the older element might not be the better choice.

parsing: $a, b, ab, ab, a, a, aa, aa, b, b, aa, a$;

dictionary (step 9): a, b, aab, aa, aaa

dictionary (step 10): a, b, aab, aa, bb

dictionary (step 11): a, b, ba, aa, bb

dictionary (step 11): a, b, ba, baa, bb

dictionary (step 12): a, b, ba, baa, aa

dictionary (step 12): a, b, ba, aa, aaa

coding: 1, 2, 3, 3, 1, 1, 4, 4, 2, 2, 4, 1

Figure 3: The AP-LRU heuristic on the extended example.

We present, now, a relaxed version of LRU. The relaxed version (RLRU) of the LRU heuristic is:

> RLRU: When the dictionary is not full, label the i^{th} element added to the dictionary with the integer $\lceil i \cdot p/k \rceil$, where k is the dictionary size minus the alphabet size and $p < k$ is the number of labels. When the dictionary is full, label the $i - th$ element with p if $\lceil i \cdot p/k \rceil = \lceil (i-1)p/k \rceil$. If $\lceil i \cdot p/k \rceil > \lceil (i-1)p/k \rceil$, decrease by 1 all the labels greater or equal to 2. Then, label the $i - th$ element with p. Finally, remove one of the elements represented by a leaf with the smallest label.

In other words, RLRU works with a partition of the dictionary in p classes, sorted somehow in a fashion according to the order of insertion of the elements in the dictionary, and an arbitrary element from the oldest class with removable elements is deleted when a new element is added. RLRU is more sophisticated than SWAP (which is the best among the simpler deletion strategies presented above) since it removes elements in a continuous way as the original LRU. In fact, we will see in section 4 that the compression performance of AP-RLRU is better than AP-SWAP. Moreover, even if it relaxes on the choice of the element to remove AP-RLRU is as good as AP-LRU.

3 The Complexity of LRU and RLRU Heuristics

The unbounded version of the LZ2 compression method is P-complete [2, 3]. This means there is a log-space reduction from any problem in P to the problem of computing LZ2 compression. Since it is believed that POLYLOGSPACE, the class of problems computed with polylogarithmic work space, is not contained in P, it is unlikely to have sublinear work space when LZ2 compression is implemented unless a deletion heuristic is applied to bound the dictionary.

The LZ2 algorithm with LRU deletion heuristic on a dictionary of size $O(\log^k n)$ can be performed in polynomial time and $O(\log^k n \log\log n)$ space (n is the length of the input string). In fact, the trie requires $O(\log^k n)$ space by using an array implementation since the number of children for each node is bounded by the alphabet cardinality. The $\log\log n$ factor is required to store the information needed for the LRU deletion heuristic since each node must have a different age, which is an integer value between 0 and the dictionary size. Obviously, this is true for any LZ2 heuristic (NC, FC, ID, AP). If the size of the dictionary is $O(\log^k n)$, the LRU strategy is log-space hard for SC^k (Steve Cook's class), the class of problems solvable simultaneously in polynomial time and $O(\log^k n)$ space [4]. The problem belongs to SC^{k+1}. This hardness result is not so relevant for the space complexity analysis since $\Omega(\log^k n)$ is an obvious lower bound to the work space needed for the computation. Much more interesting is what can be said about the parallel complexity analysis. In [4] it was shown that LZ2 compression using the LRU deletion heuristic with a dictionary of size c can be performed in parallel either in $O(\log n)$ time with $2^{O(c\log c)}n$ processors or in $2^{O(c\log c)}\log n$ time with $O(n)$ processors. This means that if the dictionary size is constant, the compression problem belongs to NC, the class of problems solvable in polylogarithmic time with a polynomial number of processors. NC and SC (the class of problems solvable simultaneously in polynomial time with polylogarithmic work space) are classes that can be viewed in some sense symmetric and are believed to be incomparable. Since log-space reductions are in NC, the compression problem cannot belong to NC when the dictionary size is polylogarithmic if NC and SC are incomparable. We want to point out that the dictionary size c figures as an exponent in the parallel complexity of the problem. This is not by accident. If we believe that SC is not included in NC, then the SC^k-hardness of the problem when c is $O(\log^k n)$ implies the exponentiation of some increasing and diverging function of c. In fact, without such exponentiation either in the number of processors or in the parallel running time, the problem would be SC^k-hard and in NC when c is $O(\log^k n)$. Observe that the P-completeness of the problem, which requires a superpolylogarithmic value for c, does not suffice to infer this exponentiation since c can figure as a multiplicative factor of the time function. Moreover, this is a unique case where somehow we use hardness results to argue that practical algorithms of a certain kind (NC in this case) do not exist because of huge multiplicative constant factors occurring in their analysis.

Finally, the LZ2 compression heuristics with RLRU deletion heuristic on a dictionary of size $O(\log^k n)$ can be performed in polynomial time and $O(\log^k n)$ space since the number of ages is constant. In fact, LZ2-RLRU compression is the first (and only so far) natural SC^k-complete problem [4].

4 Experimental Results

We show experimental results concerning the compression effectiveness of AP-RLRU with a number of classes between 2 and 6, and compare them with the results of AP-SWAP and AP-LRU. Each class is implemented with a stack. Therefore, the newest element in the class of least recently used elements is removed. Observe that if RLRU worked with only one class, after the dictionary is filled up the next element added would be immediately deleted. Therefore, RLRU would work like FREEZE. This is why we show results for a number of classes between 2 and 6. We considered natural language, programming language and postscript. The dictionary size in real life implementations has usually varied between 4,096 (twelve bits pointer size) and 65,536 (sixteen bits pointer size). In Figure 4, we present results with a dictionary size equal to 4,096.

Heuristic	English	C Programs	Postscript
LRU	.51034	.52026	.46806
RLRU2	.51193	.52039	.46971
RLRU3	.51147	.52060	.46916
RLRU4	.51153	.51957	.46902
RLRU5	.51159	.52008	.46888
RLRU6	.51150	.51982	.46919
SWAP	.68654	.71967	.61341

Figure 4: Compression ratios with dictionary size 4,096.

We experimented on samples of English text files, C programs and Postscript files. The file size varied between 100 Kilobytes and 2 Megabytes. The table shows the average of the compression ratios obtained on each sample. RLRUp denotes that the RLRU heuristic works with p classes. The compression ratios of LRU and RLRUp for $2 \leq p \leq 6$ are about the same up to the third or fourth decimal digit. On the other hand, their compression effectiveness provides about 15 to 20 percent improvement on the performance of SWAP. As mentioned in the introduction, the compression effectiveness of the RLRU heuristic is not monotonicly increasing with the value of p, which might be explained by the fact that the approach is heuristic (choosing to remove an older element is not always a better choice as discussed with the example of Figure 3).

The compression ratios of LRU and RLRU improve when the dictionary size is 65,536 as shown in Figure 5, but they compare to each other in a similar way while SWAP is only a 3 percent of LRU and RLRU on C programs and about 10 and 20 percent on English and Postscript, respectively.

5 Conclusions

We showed that a relaxed version of the best bounded size dictionary LZ2 compression technique, which uses the least recently used strategy, provides the same compression effectiveness. This version is more space efficient and easier to implement, since it

Heuristic	English	C Programs	Postscript
LRU	.32363	.38213	.33556
RLRU2	.32371	.38221	.33710
RLRU3	.32414	.38219	.33667
RLRU4	.32374	.38229	.33613
RLRU5	.32342	.38217	.33588
RLRU6	.32349	.38216	.33597
SWAP	.41402	.41657	.52827

Figure 5: Compression ratios with dictionary size 65,536.

relaxes by making a bipartition of the dictionary which defines, generally speaking, a set of less recently used elements from which one element can be removed arbitrarily.

References

[1] Bell, T.C., J.G. Cleary and I.H. Witten [1990]. *Text Compression*, Prentice Hall.

[2] De Agostino, S. [1994]. "P-complete Problems in Data Compression", *Theoretical Computer Science* **127**, 181-186.

[3] De Agostino, S. [2000]. "Erratum to P-complete Problems in Data Compression", *Theoretical Computer Science* **234**, 325-326.

[4] De Agostino, S. and R. Silvestri [2003]. "Bounded Size Dictionary Compression: SC^k-Completeness and NC Algorithms", *Information and Computation* **180**, 101-112.

[5] Storer, J.A. [1988]. *Data Compression: Methods and Theory* (Computer Science Press).

[6] Storer, J.A. [1992]. "Massively Parallel Systolic Algorithms for Real-Time Dictionary-Based Text Compression" *Image and Text Compression*, Kluwer Accademic Publishers (Storer J.A., editor), 159–178.

[7] Ziv, J. and A. Lempel [1978]. "Compression of Individual Sequences via Variable Rate Coding", *IEEE Transactions on Information Theory* **24**, 530-536.

Context-dependent Stopper encoding

Jussi Rautio

Laboratory of Information Processing Science
Helsinki University of Technology
Espoo, Finland

e-mail: Jussi.Rautio@hut.fi

Abstract. A character-based encoding method is presented for natural-language texts and genetic data. Exact string matching from the encoded text is faster than from the original text, with medium and longer patterns. A compression ratio of about 50% is achieved as a by-product. The method encodes characters with variable-length *codewords* of 2-bit *base symbols*. An advanced variant is context-dependent, using information from the previous character. The method supersedes the previous comparable methods in compression ratio, and is comparable to the best such methods in search speed.

Keywords: compressed matching, accelerator encoding, Stopper encoding

1 Introduction

As the amount of available information is constantly growing, fast information retrieval is becoming more and more important; it is a key concept in many applications, especially on-line ones. The *string matching problem* is about locating all the *occurrences* of a specific *pattern* from the *full text*. Within this paper, I will concentrate on exact string matching, requiring an exact match with the pattern and the occurrence in the full text.

A common solution to the string matching problem is to build an external *index* with pointers to the full text [15]. With an index, string matching can be done in logarithmic time. The disadvantage of these methods is an increase in space consumption. For static files, it is possible to compress the index only [15], or to compress the index separately from the full text [15]. The FM index [5] applies Burrows-Wheeler transformation [4] to the text before compression, drastically reducing the amount of necessary index data. These methods allow both an excellent compression ratio and fast string matching. However, they do not support on-line updates or approximate matching.

An alternative to indexing, it is possible to encode the full text with a local scheme specificly designed to improve search speed. A Boyer-Moore [2, 9] type string matching algorithm can be used with the encoded text, improving search speed by a constant factor. Some of these schemes even offer a significant compression ratio as a by-product. In the absence of a common term for this class of schemes, I will use the term *accelerator encoding* for all such schemes.

Although accelerator encoding falls behind compressed indexing in both compression ratio and search speed, it allows on-line updates and on-line decoding. It is suited for documents which are queried, retrieved or updated often, for example text databases or log files.

There are two types of accelerator encoding schemes: word-based and character-based. Word-based schemes [3, 12] work with whole words at a time, allowing a better compression ratio for large files but requiring a large dictionary. Their use is limited to natural-language texts where words are separated by delimiters (unlike Japanese, for example), and string matching is possible only with whole words and combinations of subsequent words. Character-based schemes [6, 13, 14] work with a fixed number of characters at a time. String matching is possible with a more free range of patterns, possibly including errors and classes of characters.

I will present a novel character-based accelerator encoding scheme and an exact string matching algorithm which works with it. Variants of the new scheme, *flexible stopper encoding*, can be used either with genetic data or with natural-language texts. The scheme is based on encoding each character of the text with a *codeword* of one or more 2-bit *base symbols*. With pure DNA code (with the alphabet `acgt`), exactly one base symbol is used for each base pair, leading to a trivial encoding. With natural-language texts, the compression ratio of the scheme is optimized with methods including context dependence of the first order. String matching from the encoded text is done with an algorithm resembling Tuned Boyer-Moore.

Compared with existing character-based schemes, the new scheme can be useful. For previous character-based schemes, the compression ratio (size of compressed file divided by size of original file) of natural-language texts was about 50% for the slowest methods and 60% for the faster ones. For my example texts, compression ratio of the new scheme is $0.3 - 2$ percentage units better than the best previous schemes, and the search speed is comparable to the fastest previous methods.

2 Background

Let $T[0, n-1]$ denote a text over the alphabet Σ. An encoding is a transformation from the text $T[0, n-1]$ to the *encoded text* $T'[0, n'-1]$, in the alphabet Σ'. String matching in an encoded text means locating all occurrences $L(P)$ of a given *pattern* $P[0, m-1]$ from the original text, with only the encoded text available. Throughout this paper, I assume that it is sufficient to locate all occurrences of the encoded pattern $L'(P')$ in the encoded text.

Accelerator encoding methods use either static or semi-static encoding. In the latter case, a small dictionary containing all necessary information for matching and decoding is saved along with the encoded text. Dynamic dictionary methods cannot be used, because the dictionary cannot be kept up to date without reading every character of the text, which would be disastrous to the performance of the algorithm.

An important property of any compression algorithm is *compression ratio*, here denoted as the size of the encoded text divided by the size of the original text, the smaller the better. For the sake of uniformity, a character of the original text is always calculated as one 8-bit byte, even with genetic data.

Byte-pair encoding (BPE) exploited the variable frequencies of consecutive character pairs. For BPE, $\Sigma' = \Sigma$. T' is a copy of T, except that the most common

pairs of consecutive characters are replaced with characters of Σ with no occurrences in T. The Manber variant [11] limits possible pairs, sacrificing compression ratio for search speed. The original variant [6] does not have this limitation, allowing a better compression ratio. Both variants have an efficient Boyer-Moore type string matching algorithm. A partially recursive version of the compression algorithm was recommended for the original variant [14], where one character in Σ' can represent one to three characters in Σ. Another variant of Byte-pair encoding called Repair [10] is even more optimized to compression ratio, but lacks an efficient search algorithm.

Our earlier comparison of these two variants [13] suggested that the Manber variant supported faster exact string matching, however its compression ratio was only 70-75% with natural-language texts. For the original variant, and especially its recursive version, the compression ratio could be as good as 45% with the same texts. However, the better the compression ratio, the slower the string matching. The scheme with the best compression ratio only allowed a slower string matching than with the original text.

Stopper encoding [13] is related to an earlier word-based method by de Moura et al. [12]. I will describe here only the 4-bit variant $\mathrm{SE}_4, 0$. The basic unit of the encoded text was the *base symbol*. It consisted of four bits, $\Sigma' = [0, 15]$. When encoding, every character of T was replaced with a corresponding *codeword*, a sequence of one or more base symbols. No codeword could be a prefix of another. More common characters were given shorter codewords than less common ones. This resembled Huffman coding. [8]

To allow faster string matching, base symbols were divided into two classes called *stoppers* and *continuers*. Let s denote the number of stoppers, such as all c in Σ' less than s are stoppers. A legal codeword $C'[0, r-1]$ consisted of zero or more symbols of the continuer class, followed by exactly one symbol of the stopper class, that is: $C'[i] \geq s$ holds for all $i < r-1$, and $C'[r-1] < s$. This made it possible to recognize codewords when starting at an arbitary location in the text, including after jumps made by a Boyer-Moore type algorithm.

The 4-bit encoded text T' was stored into the 8-bit form, two base symbols per a 8-bit computer byte, so that it could be used efficiently. String matching in the encoded text was done with a Boyer-Moore type algorithm called BM-SE, which handles whole bytes instead of individual base symbols. The algorithm operates by encoding pattern P and then locating the occurrences of the encoded pattern P' from T'. Naturally, the possible occurrences were not restricted to byte boundaries, but could start or end at either the first or the second base symbol in the byte. Because of this, two possible *alignments* of the encoded pattern must be produced by using a shift operation. The actual search algorithm was a multi-pattern version of Tuned Boyer-Moore [2, 9]. It only made one pass of the encoded text, trying to locate both of the alignments at the same time. When a presumed match was found, the preceding base symbol was checked. If it is a stopper, the match was confirmed, otherwise it was discarded.

Word-based methods resembling Stopper encoding have been used to encode whole words at a time. In schemes by de Moura et al. [12] and Brisaboa et al. [3], whole words were encoded at a time. Each was given a representation of one to three base symbols, in this case 8-bit bytes. These base symbols were divided into the continuer and stopper classes. This algorithm produced an excellent compression

ratio with natural language (currently the best one seen in accelerator encoding). De Moura's scheme used a fixed number of stoppers (128), while Brisaboa allowed free determination of the number. Search speed was only discussed by de Moura. Both schemes had the same disadvantages. They allowed matching with whole words only and could not support approximate matching. In addition, the size of the required dictionary was large compared to other methods in the field.

Our earlier comparison between Byte-pair encoding and Stopper encoding [13] suggests that Stopper encoding is superior in search speed (probably partially because of implementation issues) and that some variants of Byte-pair encoding provide a better compression ratio.

3 New solution

The new solution, *flexible stopper encoding*, is an extension to stopper encoding [13]. Stopper encoding used 4- or 6-bit base symbols depending on the variant, which had theoretical limits for the compression ratio at 50%, and 75%, respectively. Flexible stopper encoding uses 2-bit base symbols, and its theoretical limit for compression ratio is 25%. Some limitations of stopper encoding are relaxed to achieve an efficient compression ratio for this scheme. I will start by describing the basic method, and continue by discussing improvements one at a time.

Pure DNA data (of the symbols `acgt` only) can be encoded with a trivial encoding. The alphabet has only four different characters, so let us denote $\Sigma' = \{0, 1, 2, 3\}$. This means 2-bit base symbols, four of which can be stored in a 8-bit byte. This encoding gives an exact compression ratio of 25%.

Stopper encoding from the previous section can also be introduced to 2-bit base symbols. A constant s, $0 < s < 4$ is determined, dividing the encoded alphabet into two classes: stoppers and continuers. According to the definition in the previous section, for a valid codeword of length r, denoted by $C'[0, r-1]$, for all $i < r-1$ holds $C'[i] \geq s$, and $C'[r-1] < s$. The more common characters are represented by shorter codewords than the less common ones.

Note that only three legal values exist for s. 0 is impossible since no stoppers means that codewords would never end, and 4 is only valid when there are four or fewer characters in the alphabet. The best compression ratio is usually achieved with $s = 2$, but generally this scheme is too strict and must be relaxed.

Flexibility introduced to the previous scheme produces Flexible stopper encoding (FSE). The base symbols are divided into two classes as before, but the definition of the classes is changed. Stoppers function as before, but continuers are replaced with flexers, base symbols that may act either as stoppers or continuers, depending on their position in the codeword. Usually flexers act as continuers near the beginning of a codeword, and as stoppers after that.

More formally, assign values to s_i, $0 < s_i < 4$, for all reasonable i. Now, a valid codeword has exactly such base symbols that $C'[i] \geq s_i$ holds for all $i < r-1$, and $C'[r-1] < s_{r-1}$. Consequently, $s = \min s_i$.

Presumed matches preceded by flexer can be confirmed by locating the first stopper symbol preceding the presumed occurrence, and decoding after that until the identity of the flexer can be confirmed.

Context dependence allows a better compression ratio than possible if all

characters are encoded separately. The context-dependent variant is called Context-dependent FSE, or CFSE. The meanings of codewords change according to the meanings of their preceding characters. This only applies to codeword allocation, not their structure. Context dependence may be implemented in conjunction with flexibility, or independently from it.

To allow on-line locating and decoding, delimiters (spaces) are fixed always to have the same encoding. It could be possible to choose any character, but the space is chosen because it appears regularly and the most often.

A separate *successor table* S_c is constructed for each different character c occurring in the text. $S_c[0]$ is fixed to the space character, and $S_c[i]$ is the i:th common non-space successor of c. In addition, a *codeword table* is constructed, containing the $|\Sigma|$ shortest valid codewords sorted by increasing length. When encoding a character $T[s]$, its index i is located from the successor table such as $S_{T[s-1]}[i] = T[s]$, and the i:th codeword from the codeword table is put in the output stream.

Encoding and decoding algorithms are straight-forward to implement. To encode, the entire text is first scanned to count relative frequencies of characters. Then, the base symbol configuration (number of stoppers s_n) is decided, and the codeword table built. Another pass of the text is required to encode the characters one by one. Finally, the save file is built, including the base symbol configuration, the successor table, and the encoded text.

The optimal number of stoppers s_i can be calculated with an exhaustive search for small values of i. After preliminary tests, I decided to test all value combinations for all $i < 4$, and to set $s_i = s_4$ for all $i \geq 4$. The best general values for s_i for natural-language texts seem to be $1, 3, 3, \ldots$. With the context-dependent variant, all preliminary tests with natural-language texts seemed to work almost optimally with the values $s_i = 2, 3, 3, \ldots$, so this value set is automatically used with this variant.

The successor table takes $O(n^2)$ space. The list of all characters in the text is saved first. Then, for each character, its successors are saved in descending order of frequency. This takes about $4k$ space with 64 different characters in the text. Improvements are possible. The data structure used by the non-context-dependent variant is the list of characters ordered by frequency.

Decoding is done by building either a single decoding tree (non-context-dependent variant) or a separate decoding tree for each preceding character. This works exactly the same way as Huffman [8] decoding.

String matching means locating an occurrence of the pattern in the text. With FSE, it is sufficient to locate an occurrence of the encoded pattern in the encoded text, preceded by a stopper symbol. In the context-dependent variant, the first character varies according to the preceding one, but its successors do not vary and are used for the search.

The exact string matching algorithm BM-CFSE is developed from the 2-bit exact string matching algorithm used with stopper encoding, BM-SE$_{6,2}$, which was in turn influenced by Tuned Boyer-Moore [9]. The algorithm is basically a multi-pattern version of Tuned Boyer-Moore, locating all four possible alignments of the encoded pattern P' in a single pass through the text. It consists of a *preprocessing phase* and a *search phase*. The search phase alternates between a *fast loop*, which quickly weeds out most locations, and a more precise *slow loop*, which is used to confirm presumed matches found by the fast loop.

The search algorithm needs two data structures to work. The slow loop uses a *multi-mask table* S, resembling the mask table of the shift-or algorithm [1]. The fast loop uses a *jump table* D constructed from the multi-mask table, resembling the occurrence heuristic jump table from Boyer-Moore type algorithms.

To construct the multi-mask table, some definitions are required. Let P' be the encoded pattern, and P'_0, P'_1, P'_2, and P'_3 its alignments (in any order). The alignments are filled with *wild card* symbols where no base symbol is available (before the beginning or after the end of the encoded pattern). Each character in the encoded text c' consists of four base symbols c'_0, c'_1, c'_2, and c'_3. The encoded characters c' and d' are said to *unify* if and only if for all a, either c'_a and d'_a are equal, or one of them is a wild card symbol $*$.

The multi-mask table is constructed with a simple rule. Let l' be such that for all i, $P'_i[l]$ is the last full character (one not containing any wild card symbols) of P'_i. Now, $S[c,i]_a = 1$ if and only if $P'_a[i]$ unifies with c, and 0 otherwise. The value of the multi-mask is now $S[c,i] = S[c,i]_0 + 2S[c,i]_1 + 4S[c,i]_2 + 8S[c,i]_3$.

Algorithm 1 Constructing the multi-mask table S

```
fill S with 0
q ← {1,2,4,8}
for c ← 0 to 256, i ← 0 to m, a ← 0 to 4 do
  if P'_a[i] unifies with c then
    S[c,i] ← S[c,i] + q[a]
```

When the multi-mask table has been constructed, making the jump table is a trivial matter. The l'th encoded character of the pattern is always the last full encoded character of each alignment. For other encoded characters in the pattern at the location i, the possible jump length is $l - i$. The jump table construction and the fast loop are direct adaptations from Tuned Boyer-Moore. After preliminary tests, I decided to use triple loop unrolling as recommended by Hume and Sunday. A md_2 step-after-match heuristic can also be used instead of direct incrementation.

Algorithm 2 Constructing the jump table D

```
fill D with l
for i ← 0 to l, c ← 0 to 256 do
  if S[c,i] ≠ 0 then
    D[S[c,i]] ← l − i
```

The slow loop of the actual search algorithm works as a mask automaton, recognizing all 4 patterns at a time. Starting from the suspectedly first encoded character of the pattern and a state variable q positive for all masks, a bitwise-or operation is repeatedly applied to the state for each character. When the state variable reaches zero, all chances of an occurrence are lost and the fast loop can be resumed. If having gone through all the characters in the suspected pattern the state variable still has one or more positive bits, the match can be confirmed by locating a stopper symbol immediately preceding the suspected pattern.

Algorithm 3 Search algorithm: text scan phase

```
copy pattern P' to end of text T[n], T[n+1], ...
s ← l
for ever do
  k ← D[T[s]]
  while k ≠ 0 do
    s ← s + k
    k ← D[T[s]]
  i ← 0; q ← 15
  while i < l and q ≠ 0 do
    q ← q bitwise-or S[T[s − l + 1 + i], i]
    i ← i + 1
  if q ≠ 0 then
    if s = n then
      end
    else
      confirm and report occurrence(s)
  s ← s + 1
```

4 Experiments

The most important properties of accelerator encoding algorithms are search speed and compression ratio, in that order. Compression and decompression times are reported in the final version.

In the experiments, FSE and CFSE are pitted against the leading uncompressed and compressed matching algorithms. As reference algorithms, I have my earlier implementations of $SE_4, 0$ and the 6-bit Stopper encoding $SE_6, 2$, Tuned Boyer-Moore by courtesy of Hume and Sunday, and BM-BPE by courtesy of Takeda. BM-BPE comes in three versions, *fast* limiting maximum compression to two original characters per encoded character, *rec* (recommended) limiting it to three, and *max* being without limitation.

I use the Canterbury Corpus version of the King James Bible for test data. I run two separate tests with separate sets of patterns. In the first test, all patterns are whole words or beginnings of words, including the space before the beginning. Using them is a common scenario, and CFSE can search them faster than other patterns. In the second test, the patterns are unrestricted. Experiments with genetic data will be included in the final version.

All experiments are run on a 650 MHz AMD Athlon machine with 384 megabytes of main memory, running Debian Linux in single-user mode. All the programs are compiled using `gcc` with maximum optimization (flag `-O6`).

In the experiment, command-line versions of all test programs, all of them performing exactly one search per execution of program, are run several times. The programs measure their own execution time by inserting calls to the C function `clock()` into the code. This clocked time includes everything except program argument parsing and reading the file from disk.

The compression ratios are shown in Table 1. CFSE provides a better compression ratio than any of the other algorithms in all these examples. Differences between it

	KJV Bible (3.86M)
BPE_{max}	47.8%
BPE_{rec}	51.0%
BPE_{fast}	56.2%
$SE_{4,0}$	58.9%
FSE	55.6%
CFSE	47.5%

Table 1: Compression ratios.

and the maximal-compression version of BPE are $0.3-2.1$ percentage units. However, it provides an over 10 percentage units better compression ratio than the generally fastest of the other algorithms, the 4-bit Stopper Encoding.

Table 2 describes the search speed from the Bible with whole words or word beginnings, and Table 3 repeats the same test with freely chosen patterns. The performance of BM-CFSE is about the same as that of BM-$SE_{4,0}$, being somewhat faster with longer patterns and somewhat slower with shorter patterns. However, it is about twice faster than the algorithms which offer a similar compression ratio, BM-BPE_{rec} and BM-BPE_{fast}. With pattern length 5 in Table 3, the poor performance of BM-CFSE is probably because of an implementation anomaly. CFSE is minimally better with whole-word patterns than with free ones.

5 Conclusions

I have presented new accelerator encoding schemes called Flexible stopper encoding FSE and the context-dependent version CFSE, and an exact string matching algorithm for them, called BM-FSE. The new schemes produce a better compression ratio than any of the the existing accelerator encoding methods for the example natural-language text. The string matching algorithm is comparable to the fastest existing methods with both uncompressed and compressed texts.

With pure genetic data, FSE reduces to a trivial encoding with a compression ratio of exactly 25%. Compression and decompression are straight-forward operations, and mapping from the encoded text to the original is trivial. FSE can be used to store

pattern length	3	4	5	6	8	12	20
TBM	99	116	131	142	159	173	193
BM-BPE_{max}	61	63	66	68	73	81	122
BM-BPE_{rec}	56	90	95	97	128	155	212
BM-BPE_{fast}	80	84	110	113	138	177	226
BM-$SE_{4,0}$	112	152	177	203	241	301	330
BM-$SE_{6,2}$	95	160	166	219	281	398	566
BM-FSE	83	123	159	184	228	289	352
BM-CFSE	100	136	165	190	246	322	399

Table 2: Search speed for KJV Bible (word beginnings only) in kB/ms.

pattern length	5	6	8	12	20
TBM	143	148	165	189	213
BM-BPE$_{\max}$	67	69	73	81	120
BM-BPE$_{\text{rec}}$	136	138	165	214	293
BM-BPE$_{\text{fast}}$	111	115	138	169	214
BM-SE$_{4,0}$	186	220	247	358	361
BM-SE$_{6,2}$	184	213	273	435	794
BM-FSE	161	196	227	309	385
BM-CFSE	67*	139	223	307	355

Table 3: Search speed for KJV Bible (free patterns) in kB/ms.

large files of pure genetic data for efficient retrieval.

With natural-language texts, CFSE is efficient because of its good compression ratio. Its worst limit is that it relies on frequent occurrences of delimiters in the text. Unlike word-based accelerator compression schemes, CFSE still allows exact string matching with any pattern, and requires a smaller dictionary.

CFSE's advantage over BPE in compression ratio comes from the fact that BPE divides text into units encoded separately from one another. CFSE, however, always encodes according to the previous character.

In search speed, BM-CFSE is similar to BM-SE. There seems to be no fundamental difference between 4-bit base symbols and 2-bit ones. BM-CFSE benefits from its compression ratio and suffers from the omission of the first character from the fast loop.

The earlier accelerator encoding schemes had trade-offs, being either good in compression ratio and bad in speed, (BPE$_{\max}$), or the other way round (SE$_{4,0}$). It can be noted that CFSE has no such trade-off, having both a superior compression ratio and an excellent search speed. The inclusion of disk read times favors it even more. The only exception is searching with short patterns (less than 5 characters), where SE$_{4,0}$ is better.

A better compression ratio could be obtained by introducing a higher order context dependence. However, there would be problems with dictionary size, and for each pattern, two first characters would become unstable instead of one, further reducing search speed. Another interesting question is how well approximate string matching could be performed with stopper encoding or CFSE.

References

[1] Baeza-Yates, R., Gonnet, G., *A new approach to text searching*, Communications of the ACM, 35(10):74–82, 1992.

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

[3] Brisaboa, N., Fariña A., Navarro, G., and Esteller, M. *(S,C)-Dense Coding: An Optimized Compression Code for Natural Language Text Databases.* Proceedings of the SPIRE conference, pages 122-136, 2003.

[4] Burrows, M. and Wheeler, D. *A block-sorting lossless data compression algorithm.* DEC SRC Research Report 124, 1994.

[5] Ferragina, P. and Manzini, G. *An experimental study of an opportunistic index.* Proceedings of the 12th ACM-SIAM Symposium of Discrete Algorithms (SODA), 2001.

[6] Gage. P. *A new algorithm for data compression.* C/C++ Users Journal, 12(2), 1994.

[7] Golomb, S. *Run-length encoding.* IEEE Transactions on Information Theory, 12(3), 1966.

[8] Huffman, D. *A method for the construction of minimum-redundancy codes.* Proceedings of the IRE 40, 1098-1101. David Applegate et al, 1952.

[9] Hume, A. and Sunday, S. *Fast string searching.* Software Practice and Experience, 21:1221–1248, 1991.

[10] Larsson, N., Moffat, A. *Offline dictionary-based compression.* Proc. IEEE, 88(11), 1722–1732, 2000.

[11] Manber, U. *A text compression scheme that allows fast searching directly in the compressed file.* In Proc. Combinatorial Pattern Matching, Lecture Notes in Computer Science, 807:113–124. Springer-Verlag, 1994.

[12] de Moura, E., Navarro, G., Ziviani, N. and Baeza-Yeates, R. *Fast and flexible word searching on compressed text.* ACM Transactions on Information Systems, 18(2):113–139, 2000.

[13] Rautio, J., Tanninen, J., and Tarhio, J. *String matching with stopper encoding and code splitting.* Proc. CPM '02, Combinatorial Pattern Matching (ed. A. Apostolico, M. Takeda), Lecture Notes in Computer Science 2373, Springer, 2002, 42-52.

[14] Shibata, Y., Matsumoto, T., Takeda, M., Shinohara, A. and Arikawa, S. *A Boyer-Moore type algorithm for compressed pattern matching.* Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching (LNCS 1848), pages 181–194. Springer-Verlag, 2000.

[15] Witten, I., Moffat, A., Bell, T. *Managing gigabytes.* Morgan Kaufmann Publishers, Academic Press, 1999.

[16] Ziv, J. and Lempel, A. *A universal algorithm for sequential data compression.* IEEE Transactions on Information Theory, 23:337–343, 1977.

General Pattern Matching on Regular Collage System

Jan Lahoda and Bořivoj Melichar

Dept. of Computer Science and Engineering
Faculty of Electrical Engineering
Czech Technical University
Karlovo náměstí 13
121 35 Prague 2
Czech Republic

e-mail: {lahodaj,melichar}@fel.cvut.cz

Abstract. This paper presents a brand new approach to the general pattern matching on regular collage systems. Our approach provides $\mathcal{O}(||D|| + |S| + E)$ (where E is the preprocessing cost) worst-case time complexity. It is based on fact that a deterministic finite automaton is able to distinguish only a limited number of strings.

Keywords: pattern matching in compressed text, pattern matching, text compression, finite automata, collage systems

1 Introduction

In the concurrent world, each year more and more data are to be stored and processed. It also seems that the amount of data to be stored and processed grows faster than the data storage media and processing appliances.

The pattern matching in compressed text helps in both directions: the data are compressed in order to consume less space, and then an algorithm for the pattern matching in compressed text is employed to simplify the data processing.

In this paper, we provide a new approach for general (regular expression) pattern matching over collage systems. Collage systems are means of representing several compression methods in a unique way, and our approach uses finite automata as unique approach for solving many pattern matching problems.

2 Basic notions and notations

Let us denote *pref*(P), *fact*(P) and *suff*(P) set of all prefixes, factors and suffixes (respectivelly) of string P. Let us denote *LSpref*(w, P) and *LSfact*(w, P) longest suffix of w that is concurrently a prefix (factor) of P.

2.1 Collage systems

Collage systems [KST+99, KMT+01] are means of representing several compression methods in a unique way. A collage system is a pair (D, S), where:

D (called dictionary) is a sequence of assignments in form $X_1 = expr_1$; $X_2 = expr_2$; $\cdots$; $X_l = expr_l$ where the expression for assignment X_k is constructed in one of these forms:

a	for any $a \in (A \cup \{\varepsilon\})$,	*(primitive assigment)*
X_iX_j	for $i, j < k$,	*(concatenation)*
${}^{[j]}X_i$	for $i < k$ and an integer j,	*(prefix truncation)*
$X_i^{[j]}$	for $i < k$ and an integer j,	*(suffix truncation)*
$(X_i)^j$	for $i < k$ and an integer j.	*(j times repetition)*

S (called sequence) is a sequence of assignments defined in D in form

$$S = X_{i_1}, X_{i_2}, \ldots, X_{i_n}$$

Let us denote u_i string representing assigment X_i and $u = u_{i_1}u_{i_2}\ldots u_{i_n}$ string representing the collage system.

[KST+99] describes how to express various compression methods using collage systems.

Several types of collage systems were defined in [KST+99]. The two most important types in our case are regular and simple collage systems. The dictionary of a regular collage system can contain assignments only in form of a or X_iX_j. The simple collage systems are such regular collage systems, where for each assigment in form of X_iX_j holds either $X_i = a$ or $X_j = a$ for some $a \in A$.

For example, let us consider the following collage system: $D = \{X_1 = a, X_2 = b, X_3 = X_1X_2, X_4 = X_3X_3\}$, $S = \{X_4X_4\}$. Then the assignments represent the following strings $u_1 = a$, $u_2 = b$, $u_3 = ab$, $u_4 = abab$ and the whole collage system represents string $u = abababab$.

3 Previous work

The collage systems were defined in [KST+99] as generalization of several compression methods. In this paper, an algorithm for exact one pattern matching was provided. In [KMT+01], an algorithm for exact multiple pattern has been provided. For a given collage system and pattern(s) of total length m, these algorithms have time complexity $\mathcal{O}(||D|| + |S| + m^2 + r)$ and space complexity $\mathcal{O}(||D|| + m^2)$.

4 Main result

In this section we will present algorithm for pattern matching on collage systems. In order to make the pattern matching algorithm run in time $\mathcal{O}(||D|| + |S| + \tau)$ (where τ represents preprocessing time) it is necessary to use only $\mathcal{O}(1)$ time for each item in the sequence S and each expression in dictionary D. Therefore, in this section we present a new approach to compute "descriptions" of each $X_i \in D$ so we are able to

create and update description for each X_i in $\mathcal{O}(1)$ time and to process each item in sequence S in $\mathcal{O}(1)$ time.

Our solution is based on the fact that there is only a very limited number of strings that behave "differently" in a given pattern matching automaton (see Definition 1). So the main idea is to find a (shorter) string (so called representant string), from a limited set of predefined strings, for each dictionary item, that will behave in the same way in the pattern matching automaton. As will be shown later, the representant string of concatenation of two representant strings of two dictionary items can then be computed in $\mathcal{O}(1)$.

Definition 1. *Let $M = (Q, A, \delta, q_0, F)$ is a pattern matching automaton (deterministic finite automaton) and δ^* is a transitive reflexive closure of the transition function δ. Then relation $\sim_M$ (shortcut $\sim$ will be used in the future when the automaton M is clear from the context) is defined as follows: for each two strings $u, v \in A^*$ holds that $u \sim v$ if and only if for each $q \in Q$ holds:*

1. *$\delta^*(q, u) = \delta^*(q, v)$*
2. *exactly one of the following is true:*
   - *there exist strings $u' \in A^*$, $u' \in pref(u)$ and $v' \in A^*$, $v' \in pref(v)$ such that $\delta^*(q, u') \in F$ and $\delta^*(q, v') \in F$,*
   - *for all strings $u' \in A^*$, $u' \in pref(u)$ and $v' \in A^*$, $v' \in pref(v)$ holds that $\delta^*(q, u') \notin F$ and $\delta^*(q, v') \notin F$,*

Definition 2. *For a given deterministic finite automaton M, let us suppose that an ordering is given on the set of states Q, so we can enumerate the states in an order.*

For each string $u \in A^*$, let us define signature $\mathcal{S}(u)$ as a vector of pairs $\mathcal{S}(u) = ((q'_1, f_1), \ldots, (q'_{|Q|}, f_{|Q|}))$ of length $|Q|$, where a pair (q'_i, f'_i), $q'_i \in Q$ and $f_i \in \{true, false\}$. On position i is computed as $q'_i = \delta^*(q_i, u)$ and boolean f_i is true if and only if there is a prefix u' of u such that $\delta^*(q_i, u') \in F$, false otherwise.

Example 3. *Let us consider finite deterministic automaton shown in Figure 1. For order of states: (A, B, C), the signatures are as follows:*

u	$\mathcal{S}(u)$		
	A	B	C
ε	(A, f)	(B, f)	(C, f)
a	(B, t)	(B, f)	(B, f)
ab	(C, t)	(C, t)	(C, t)
aba	(B, t)	(B, t)	(B, t)
abc	(A, t)	(A, t)	(A, t)
b	(A, t)	(C, t)	(A, f)
ba	(B, t)	(B, t)	(B, f)
bc	(A, t)	(A, t)	(A, f)
c	(A, f)	(A, f)	(A, f)

Theorem 4. *For a given deterministic finite automaton M and for each two strings $u, v \in A^*$ holds that $u \sim v$ if and only if $\mathcal{S}(u) = \mathcal{S}(v)$.*

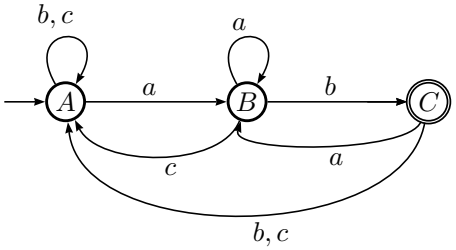


Figure 1: Example pattern matching automaton

Proof. Leads directly from Definitions 1 and 2. □

Example 5. *Let us consider finite deterministic automaton shown in Figure 1 and the order of states:* (A, B, C). *Then the signature of strings aba and cccaba is the same:* $\mathcal{S}(aba) = \mathcal{S}(cccaba) = ((B,t),(B,t),(B,t))$. *Therefore it holds that aba* $\sim$ *cccaba.*

Theorem 6. *The relation* $\sim$ *is equivalence, and moreover, it is right congruence.*

Proof. To prove that relation $\sim$ is an equivalence, we need to prove that it is reflexive, symmetric and transitive (for each $u, v, w \in A^*$):

reflexivity it is obvious that $\mathcal{S}(u) = \mathcal{S}(u)$,

symmetry if $u \sim v$, then $\mathcal{S}(u) = \mathcal{S}(v)$, and also $\mathcal{S}(v) = \mathcal{S}(u)$, and therefore $v \sim u$,

transitivity if $u \sim v$ and $v \sim w$, then $\mathcal{S}(u) = \mathcal{S}(v)$ and $\mathcal{S}(v) = \mathcal{S}(w)$, then $\mathcal{S}(u) = \mathcal{S}(w)$ and therefore $u \sim w$.

To prove that relation $\sim$ is a right congruence, it is necessary to prove that for all $\alpha, \beta, \gamma \in A^*$, such that $\alpha \sim \beta$ holds $\alpha\gamma \sim \beta\gamma$. Let us prove this by contradiction: let us suppose there is an automaton M, and strings $\alpha, \beta, \gamma \in A^*$, such that $\alpha \sim \beta$, but not $\alpha\gamma \sim \beta\gamma$. This means that there exists a state $q \in Q$ such that at least one of there is true:

1. $\delta(q, \alpha) = \delta(q, \beta)$, but $\delta(q, \alpha\gamma) \neq \delta(q, \beta\gamma)$,

2. no $\alpha' \in$ *pref*(α) and $\beta' \in$ *pref*(β) exists such that $\delta(q, \alpha') \in F$ and $\delta(q, \beta') \in F$ and there exists $(\alpha\gamma)' \in$ *pref*$(\alpha\gamma)$ such that $\delta(q, (\alpha\gamma)') \in F$ and there is no $(\beta\gamma)' \in$ *pref*$(\beta\gamma)$ such that $\delta(q, (\beta\gamma)') \in F$ (or vice versa).

The first variant is not possible, because $\delta(q, \alpha) = \delta(q, \beta) = q'$ and $\delta(q', \gamma) = q''$, and therefore $q'' = \delta(q, \alpha\gamma) = \delta(q, \beta\gamma)$.

The second variant is not possible, because obviously: $\delta(q, \alpha) = \delta(q, \beta) = q'$, $|(\alpha\gamma)'| > |\alpha|$ and therefore there must be $\alpha\gamma' = (\alpha\gamma)'$ and so if $\delta(q, \alpha\gamma') \in F$ then $\delta(q', \gamma') \in F$ and $\delta(q, \beta\gamma') \in F$, and so exists $\beta\gamma' \in$ *pref*$(\beta\gamma)$ which breaks this condition.

Therefore, such automaton M and string α, β, γ cannot exist, and therefore the equivalence $\sim$ is a right congruence. □

The equivalence defined in the Definition 1 defines strings that behave "in the same way" in the pattern matching automaton (they lead for a given state $q \in Q$ into the same state q' and they remember whether or not they passed through a final state).

Definition 7. *Let $W \subset A^*$ be a set of class representatives for partition $A^*/\sim$ of A^*, such that for each $w \in W$ holds that there does not exist any w' such that $w \sim w'$ and $|w'| < |w|$.*

Theorem 8. *For a given automaton $M = (Q, A, \delta, q_0, F)$, the corresponding set W is finite and moreover has at most $(2|Q|)^{|Q|}$ elements.*

Proof. As for each pair of strings $u, v \in A^*$ holds that $u \sim v$ if and only if $\mathcal{S}(u) = \mathcal{S}(v)$, it is therefore clear that there cannot be more classes of equivalence in partition $A^*/\sim$ than is the number of distinct vectors. The number of different vectors is $(2|Q|)^{|Q|}$ for a given automaton (each tuple of the vector can contain $2|Q|$ distinct values, and $|Q|$ tuples are independently combined into a vector). □

Although the size of set of representatives W is overwhelming, for most practical purposes the size of this set is much smaller. Section 5 analyses these cases.

Algorithm 4 shows how to construct the set of representatives W for a given automaton M.

Algorithm 4 Construction of the set of representatives W

Require: Deterministic finite automaton M
Ensure: Set of representatives W correspoding to the automaton M

```
1: U = {ε}
2: while U is not empty do
3:     remove a w from U such that there is no w' ∈ U such that |w| < |w'|.
4:     if S(w) is not in S_M then
5:         W = W ∪ {w}
6:         for each a ∈ A put wa into U
7:         S_M = S_M ∪ {S(w)}
8:     end if
9: end while
```

To solve the pattern matching problem on the (regular) collage system in $\mathcal{O}(||D|| + |S|)$ time, it is necessary to compute $w \in W$ corresponding to each item in $\mathcal{O}(1)$ time. For the simple assignment (like $X_i = a$), it is trivial. In order to compute representant string for $X_k = X_i X_j$ from representant strings of X_i and X_j, a characteristic automaton M_H is defined.

Definition 9. *For a given deterministic finite automaton M and correspoding set of representatives W,* characteristic *automaton $M_H = (Q_H, W, \delta_H, q_{H0}, \emptyset)$ is defined in the following way:*

Q_H: $Q_H = W$,

$\delta_H : Q_H \times W \to Q_H$: $\delta_H(q_H, w) = u$, *where $w, u \in W$ such that $q_H w \sim u$,*

q_{H0}: $q_{H0} = \varepsilon$.

The automaton M_H has obviously space complexity $\mathcal{O}(|W|^2)$ for regular collage systems. For simple collage systems, simplified characteristic automaton can be employed which space complexity is only $\mathcal{O}(|W||A|)$ (only expressions in form $X_k = X_i a$ or $X_k = aX_i$, where $a \in A$ are allowed).

Another problem to solve is the match detection. This can be done using a special final markers table $\mathcal{F}$.

Definition 10. *For a given deterministic finite automaton M and correspoding set of representatives W, the final markers table $\mathcal{F}$ is defined for each $w \in W$ and $q \in Q$ such that $\mathcal{F}[w, q] = true$ if and only if there exists a $w' \in pref(w)$ such that $\delta(q, w') \in F$.*

Algorithm 5 Pattern Matching on Regular Collage Systems

▷ Preprocessing phase

for the given pattern P and pattern matching problem $\mathcal{P}$ construct pattern matching automaton M, characteristic automaton M_H and final markers table $\mathcal{F}$.
compute representative for each dictionary item from the dictionary D

▷ Pattern matching phase

$q = q_0$
$j = 0$
for all X from $S = \{X_{i_1}, X_{i_2}, \ldots, X_{i_n}\}$ **do**
    let $w \in W$ is the representant string corresponding to X
    if $\mathcal{F}[w, q]$ is true **then**
        report occurence(s) between positions j and $j + |X.u|$
    end if
    $q = \delta(q, w)$
    $j = j + |X.u|$
end for

5 On the Size Of W

Although the worst-case size of the set of representatives W for a given automaton M is overwhelming (up to $(2|Q|)^{|Q|}$), for many practical cases the size of this set is much smaller. In this section, a proof that for exact one pattern matching of an aperiodic pattern of length m, the size of the set W is $\mathcal{O}(m^2)$. Moreover, results of pratical experiments for commonly used patterns and pattern matching problem are discussed.

5.1 Exact One Pattern Matching

In this section, we prove that for each deterministic finite automaton constructed to solve exact one pattern matching for an aperiodic pattern of length m (see Definition 14), the size of set W is $\mathcal{O}(m^2)$.

Moreover, our experiments have shown, that the aperiodic pattern is the worst-case with regard to the size of set W. We have created automata and sets W for all patterns of length 6, and none of these patterns performed worse than the aperiodic pattern.

Definition 11. *Let automaton $M = (Q, A, \delta, q_0, F)$ be a pattern matching automaton for exact one pattern matching of pattern P.*

Then for each state $q \in Q$ exists exactly one string $u \in A^*$ such that $u \in pref(P)$, $\delta(q_o, u) = q$ and there is no shorter prefix with the same property. Let us define function corr, which for each state q has value of the appropriate string u.

Lemma 12. *For a given automaton M, let us define set W' which fulfills these properties:*

1. *$\varepsilon \in W'$*
2. *for each $a \in A$ and $w' \in W'$ exists $u' \in W'$ such that $u' \sim w'a$*

Then for each $w \in A^*$ exists a $w' \in W'$ such that $w \sim w'$.

Proof. (by induction by the length of w)

1. For $w = \varepsilon$, $w \in A^*$, there clearly exists $w' = \varepsilon$ (the first condition on set W'), such that $w \sim w'$.
2. Let us suppose that the claim holds for all $w \in A^*$, $|w| \leq k$. Than for each $a \in A$ and $w \in W$, $|w| \leq k$ holds: there exists $w' \in W'$ such that $w \sim w'$. There also exists $u' \in W'$ such that $u' \sim w'a$. As the equivalence $\sim$ is a right congruence, it also holds that $wa \sim u'$. The claim therefore also holds for all $|wa| \leq k + 1$.

□

Corollary 13. *For set W' defined in Lemma 12 holds that $|W| \leq |W'|$.*

Proof. (by contradiction)Let us suppose there exists such automaton M, correspoding set of class representatives W and a set W' such that $|W| > |W'|$. But then there must be two $w_1, w_2 \in W$, $w_1 \neq w_2$ such that there exists $w' \in W'$, $w_1 \sim w'$, $w_2 \sim w'$. As $\sim$ is equivalence, it is clear that $w_1 \sim w_2$, and that means that strings w_1 and w_2 are in the same class of equivalence, and therefore set W is not set of class representatives, which is the contrandiction with the assumptions. Therefore such automaton M and sets W and W' cannot exist. □

Definition 14. *Let us call pattern $P = a_1 a_2 \dots a_m$ of length m such that for each two $i, j \in < 1, m >$, $i \neq j$ holds $a_i \neq a_j$ aperiodic.*

Lemma 15. *For an aperiodic pattern $P = a_1 a_2 \cdots a_m$ of length m, alphabet $A = a_1, \dots, a_m, x$, and corresponding automaton M, construct set $W' = \{w' : w' \in fact(P)$ or $w' = sxp, s \in suff(P), p \in pref(P)\}$. The set W' fulfills requirements defined in Lemma 12.*

Proof. As $\varepsilon \in fact(P)$, it is clear that $\varepsilon \in W'$.

Let as for each factor $f \in fact(P)$ denote a_n the symbol in A for which $fa \in fact(P)$. Note that there is no a_n for all $f \in suff(P)$.

For each $w' \in W'$ and $a \in A$, let us analyse all possibilities (for each combination of w' and a, only the topmost step is valid):

- $w' = \varepsilon$: it clearly holds $a \in \mathit{fact}(P)$ or $a = x$, and so $a \in W'$
- $w' \in \mathit{fact}(P)$, $a = a_n$: it clearly holds $fa \in \mathit{fact}(P)$ and so $fa \in W'$
- $w' \in \mathit{fact}(P)$, $w' \notin \mathit{suff}(P)$, $a = a_1$: as for each state $q \in Q$ holds that $\delta(q, a_1) = q_1$, and that there is no such $q \in Q$ and $w'' \in \mathit{pref}(w')$ such that $\delta(q, w'') \in F$, it holds that $w'a \sim a_1$.
- $w' \in \mathit{fact}(P)$, $w' \notin \mathit{suff}(P)$, $a \neq a_1$, $a \neq a_n$: as the pattern is not periodic, the longest suffix of $w'a$ that is prefix of P is ε, and that there is no such $q \in Q$ and $w'' \in \mathit{pref}(w')$ such that $\delta(q, w'') \in F$, it holds that $w'a \sim \varepsilon$.
- $w' = P$, $a = a_1$: as for each state $q \in Q$ holds that $\delta(q, P) = q_m \in F$, $\delta(q_m, a_1) = q_1$, it holds that $Pa \sim Pxa$.
- $w' = P$, $a \neq a_1$: as for each state $q \in Q$ holds that $\delta(q, P) = q_m \in F$, $\delta(q_m, a) = q_0$, it holds that $Pa \sim Px$.
- $w' \in \mathit{suff}(P)$, $a = a_1$: as for one state $q_s \in Q$ holds that $\delta(q_s, w') = q_m$, and for all other $q \in Q$ holds that $\delta(q, w') = q_0$, it holds that $w'a \sim w'xa_1$.
- $w' \in \mathit{suff}(P)$, $a \neq a_1$: as for one state $q_s \in Q$ holds that $\delta(q_s, w') = q_m$, and for all other $q \in Q$ holds that $\delta(q, w') = q_0$, it holds that $w'a \sim w'x$.
- $w' = sxp$ for some $s \in \mathit{suff}(P)$, $p \in \mathit{pref}(P)$, $a = a_n$ (a_n regarding prefix p), $pa_n \neq P$: it clearly holds that: $w'a \in W'$.
- $w' = sxp$ for some $s \in \mathit{suff}(P)$, $p \in \mathit{pref}(P)$, $a = a_n$ (a_n regarding prefix p), $pa_n = P$: it clearly holds that: $w'a \sim P$.
- $w' = sxp$ for some $s \in \mathit{suff}(P)$, $p \in \mathit{pref}(P)$, $a = a_1$, $p \neq P$: as $\delta^*(q, pa_1) = q_1$ for all $q \in Q$, and there is no $p'' \in \mathit{pref}(p)$ such that $\delta(q, p'') \in F$, it holds that $w'a \sim sxa_1$.
- $w' = sxp$ for some $s \in \mathit{suff}(P)$, $p \in \mathit{pref}(P)$, $a = a_1$, $p = P$: as $\delta^*(q, Pa_1) = q_1$ for all $q \in Q$, and $\delta(q, P) \in F$, it holds that $w'a \sim Pxa_1$.
- $w' = sxp$ for some $s \in \mathit{suff}(P)$, $p \in \mathit{pref}(P)$, $p \neq P$: as $\delta^*(q, pa) = q_0$ for all $q \in Q$, and there is no $p'' \in \mathit{pref}(p)$ such that $\delta(q, p'') \in F$, it holds that $w'a \sim sx$.
- $w' = sxp$ for some $s \in \mathit{suff}(P)$, $p \in \mathit{pref}(P)$, $p = P$: as $\delta^*(q, Pa) = q_0$ for all $q \in Q$, and $\delta(q, P) \in F$, it holds that $w'a \sim Px$.

□

Lemma 16. *For an aperiodic pattern $P = a_1a_2 \cdots a_m$ of length m, alphabet $A = a_1, \ldots, a_m, x$, and corresponding automaton M for exact one pattern matching, the set W has at most $\mathcal{O}(m^2)$ items.*

Proof. As the set defined in the Lemma 15 has at most $\mathcal{O}(|Q|^2)$ elements, and according to Corollary 13, the set W has at most $\mathcal{O}(|Q|^2)$ elements.

As proven in [Hol00], the automaton for exact one pattern matching of pattern of length m has $m + 1$ states ($|Q| = m + 1$) and therefore $|W| = \mathcal{O}(m^2)$. □

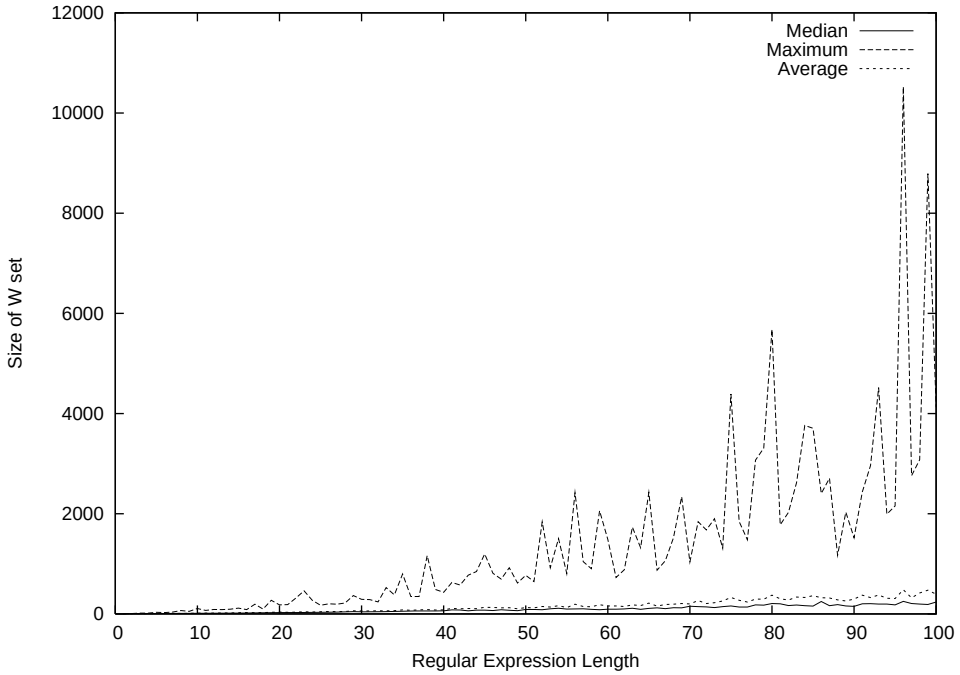


Figure 2: Size of set of representants W for a random regular expression of given length

5.2 Regular Expressions

We have constructed 100 random regular expressions for lengths 1 to 100 (therefore we tested 10000 regular expressions). We define the length of the regular expression as the number of symbols from the alphabet in the regular expression, so operators and brackets are not counted into the length of the regular expression. The regular expressions were prepended with ".*" to simulate pattern matching algorithms. The results from these experiments are summarised in Figure 2.

As can be seen from the graph, the size of the set W for our regular expressions grows much less than $|Q|^{|Q|}$ (note that $Q = \mathcal{O}(2^m)$ where m is the length of the regular expression). Therefore, it seems that the proposed algorithm may be usefull for a wide range of practical applications.

6 Conclusion

In this paper, a new method for general pattern matching on collage systems is presented. This method allows general pattern matching on the regular collage systems in linear time with respect to the size of the collage system.

Although the preprocessing time and space requirements of this method may be very high, in Section 5 is shown that for some practical applications the requirements are more acceptable. Moreover, it is possible to used here-presented approach as long as the preprocessing requirements are acceptable (gaining very fast processing time) and resort to another algorithm (decompress&search in the worst-case) otherwise.

References

[Hol00] J. Holub. *Simulation of Nondeterministic Finite Automata in Pattern Matching*. PhD thesis, Faculty of Electrical Engineering, Czech Technical University, Prague, Czech Republic, 2000.

[KMT+01] Takuya Kida, Tetsuya Matsumoto, Masayuki Takeda, Ayumi Shinohara, and Setsuo Arikawa. Multiple pattern matching algorithms on collage system. *Lecture Notes in Computer Science*, 2089:193–206, 2001.

[KST+99] T. Kida, Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa. A unifying framework for compressed pattern matching. In *procString Processing and Information Retrieval: A South American Symposium'99*, pages 89–96. IEEE CS Press, 1999.

Alphabets in Generic Programming

Juha Kärkkäinen

Department of Computer Science, P.O.Box 68 (Gustaf Hällströmin katu 2 B)
FI-00014, University of Helsinki, Finland

e-mail: Juha.Karkkainen@cs.helsinki.fi

Abstract. We initiate the design of a software library of algorithms and data structures on strings. The design is based on generic programming, which aims for a single implementation of an abstract algorithm that works in every situation, particularly with any kind of string or sequence, without any disadvantage to a more specific implementation. The design requires a deep understanding of both different algorithms and various types of strings. In this paper, we address one aspect of strings, the alphabet. The main contribution is a novel definition of the concept of an alphabet in a program. The key feature is the recognition of two levels, the level of abstract algorithms and the level of concrete programs, and the establishment of a connection between the levels. Based on the definition, we provide a sketch of a design for alphabet traits, a crucial abstraction layer between algorithms and strings.

1 Introduction

Algorithms and data structures on strings [5, 12] are often practical: implementable with a reasonable effort and usable for real world problems. Indeed, many basic algorithms have been implemented several times in applications or for experimental evaluation, and practical aspects have been an important area of research (see, for example, [11]). However, existing implementations are usually hard to find, of low quality (even incorrect), or difficult to modify for new purposes. Thus, someone needing an implementation faces a lot of work whether implementing from scratch or starting from an existing implementation.

A good software library can significantly ease the task of an implementer as it provides a single source of high quality, well-tested and flexible implementations of algorithms and data structures. There are successful libraries in several areas of algorithmics including fundamental algorithms and data structures (STL [3]), graph algorithms (LEDA [10]), and computational geometry (CGAL [8]). Stringology has been identified as another area that is ripe for a software library and a proposal has been made [7], but nothing comparable to STL, LEDA or CGAL exists, yet.

The purpose of this paper is to initiate the design for a software library of algorithms and data structures on strings. The library design is based on the generic programming paradigm [3], which was established by STL and is also the basis of CGAL. Generic programming strives for simultaneous flexibility and efficiency through implementations that work with as many data types as possible without a loss of efficiency. Ideally, one can use a single generic implementation of an abstract algorithm in every situation without any disadvantage to a specialized implementation. In the case

of stringology, generic programming means that the library algorithms should work efficiently with almost any kind of a string or a sequence.

Generic programming achieves its goal of genericity by the means of an abstraction layer between algorithms and the data they operate on, in this case strings. Designing this layer is the crucial step in designing an algorithm library. The layer needs to operate with a large number of different algorithms and a wide variety of string types, and a good design must be based on a deep understanding of both. Full analysis is far beyond the scope of this paper but we will start with one fundamental aspect.

A string can be defined as a sequence of characters, which reveals the two largely orthogonal aspects of strings: the *sequence aspect* and the aspect of individual character, which we will call the *alphabet aspect*. Sequences are central to STL, and there, a deep analysis of sequences and algorithms on sequences has led to the concept of iterators. A good introduction to iterators can be found in [3]. For understanding this paper, it is enough to think iterators as pointers to an array, with a sequence represented by a pair of iterators indicating the beginning and the end of the sequence.

We will concentrate on the alphabet aspect. We start with a motivating example of a simple algorithm illustrating the problem of alphabets in generic programming. We will then go on to analyze and define the concept of an alphabet. The central feature is the recognition of two levels, the level of abstract algorithm design and analysis, and the level of concrete implementations and programs. We establish a formal connection between the levels enabling one to see an alphabet at both levels simultaneously. Finally, we sketch the design of *alphabet traits* that forms a part of the abstraction layer between algorithms and strings.

C++ is the language of LEDA, STL and CGAL, and has the best support for generic programming (out of widely used languages, at least). It is thus the obvious choice of language. The fast development of template metaprogramming techniques in recent years [1, 2, 6, 13] has brought us closer to achieving the ideals of generic programming. Understanding this paper does not require knowledge of these techniques, though some knowledge of C++ may be helpful.

2 Example Algorithm

Consider the following simple algorithm that computes the number of distinct characters in a string.

```
count_distinct(string S)
1   seen := ∅
2   for each character c of S do
3       seen := seen ∪ {c}
4   return |seen|
```

Two points in this algorithm are problematic for a generic implementation. One is the set *seen*, and the other is the iteration over the characters of S. The latter is involved with the sequence aspect of the string and is the kind of thing that iterators were designed for. The former is involved with the alphabet aspect and could be handled using the generic set data structure in STL. This would lead to the following

typical STL-style function:[1]

```
template <typename Iterator>
int count_distinct(Iterator begin, Iterator end)  {
  typedef iterator_traits<Iterator>::value_type chartype;
  set<chartype> seen;
  for (Iterator i = begin; i != end; ++i)
    seen.insert(*i);
  return seen.size();
}
```

This is a quite generic implementation, but it is slower than necessary in many cases since the set is implemented with a balanced search tree. In particular, in the most common case of the characters being of type `char`, the following function is significantly faster for a long string.

```
template <typename Iterator>
int count_distinct(Iterator begin, Iterator end)  {
  vector<bool> seen(256,false);
  for (Iterator i = begin; i != end; ++i)
    seen[*i]=true;
  return count(seen.begin(), seen.end(), true);
}
```

Using standard techniques, we could use the latter implementation, when the characters are of type `char` and the former otherwise. However, choosing the optimal data structure for the set is not that simple:

- If the alphabet is a small range of integers, we should use a vector, whatever the character type.
- If the alphabet is a small set of integers from a large range, a hash table might be the choice.
- Even balanced tree is not quite as generic as is possible. It requires order comparisons, which not all C++ types have, and which, even when available, might do the wrong thing (see below). In such cases, we could still implement the set as an unordered list.

Further complexity can be created by an unusual concept of character equality. Consider the following examples:

- With a case insensitive alphabet, an upper case and a lower case letter are considered to be the same character, and are counted as one.

[1] We have simplified the C++ code in this paper by ignoring some quirks of C++: omitted `typename` at places, used `vector<bool>` though it's not the best choice, assumed `char` is unsigned, etc.

- A character in a protein sequence might contain information about secondary or tertiary structure in addition to the amino acid. If we want to count distinct amino acids, however, the extra information should be ignored when comparing characters.
- Two floating point values might be considered the same if they round to the same integer.

All the examples could be handled by creating first a new string using an appropriate character conversion, but at the cost of a time and space overhead. In character counting, the overhead is probably small, but in other cases it could be significant. For example, the Boyer–Moore algorithm [4] usually accesses only a small fraction of characters and converting all of them could be costly.

The above discussion shows that we cannot expect the C++ type of characters to carry all relevant information about the alphabet. A separate entity (a type or an object) is needed for that purpose. In generic programming, such entities are known as *traits* (see `iterator_traits` above). The C++ standard library does, in fact, include something called character traits, but they are more of a relic from time before generic programming. We will call our traits *alphabet traits*.

Let us finally see what an implementation of our counting function using alphabet traits might look like. (A full implementation with a usage example is in the Appendix.)

```
template <typename Iterator, typename Alphabet>
int count_distinct(Iterator begin, Iterator end, Alphabet A)  {
  typedef generate_set<Alphabet>::type charset;
  charset seen(A);
  for (Iterator i = begin; i != end; ++i)
    seen.insert(*i);
  return seen.size();
}
```

Here `Alphabet` is an alphabet traits *type* and `A` an alphabet traits *object*. The *metafunction* `generate_set` chooses the appropriate implementation for the set.

Despite its simplicity, the above algorithm captures a lot of the difficulties with alphabets in generic programming. For example, the problem of implementing a node in a trie or an automaton is closely related to the problem of implementing the set *seen*.

3 Alphabet

Alphabet traits describe the properties of an alphabet, which itself is a more abstract entity. Before designing alphabet traits, we need to define more clearly what an alphabet is. That is the purpose of this section and, indeed, the main purpose of this paper.

When we talk about an alphabet in a generic implementation of an abstract algorithm, we are talking about two different things. One is the *abtract alphabet*, the mathematical set appearing in problem definitions, abstract algorithms and their

asymptotic analysis. The other is the *concrete alphabet*, which is a specific representation of an alphabet in a program.

3.1 Abstract Alphabet

An abstract alphabet is the set of all possible characters. The following properties of the set are of interest:

- *ordering*: Does the alphabet have a linear order?
- size: Is it *constant*, σ (*finite*), or *infinite* (unknown)?
- *integrality*: Are the character integers?

One could also specify other properties but these are sufficient for most situations arising in design and analysis of abstract algorithms. Note that we allow infinite and unordered alphabets.

Consider the character counting algorithm from Section 2. The best implementation of the character set and the resulting complexity depend on the properties of the alphabet. For a string of length n, we have the following complexities for various kinds on alphabets:

- infinite: $\mathcal{O}(n^2)$
- finite: $\mathcal{O}(n \min\{n, \sigma\})$
- constant: $\mathcal{O}(n)$
- ordered: $\mathcal{O}(n \log n)$
- finite and ordered: $\mathcal{O}(n \log \min\{n, \sigma\})$
- finite and integral: $\mathcal{O}(n + \sigma)$ deterministic, $\mathcal{O}(n)$ randomized

3.2 Concrete Alphabet

A concrete alphabet is a representation of an abstract alphabet based on the following three principles:

- All character representations are values of a single C++ type `T`.
- Not all values of `T` need to represent a character.
- Multiple values may represent the same character.

Formally, a *concrete alphabet* $\mathcal{A}$ is a triple $(\mathtt{T}, C, \sim)$, where

- `T` is a C++ type.
- C is a subset of the possible values of the type `T`.
- $\sim$ is an equivalence relation on C.

The concrete alphabet $\mathcal{A}$ defines an *abstract alphabet* $\widetilde{\mathcal{A}}$ as the set of equivalence classes of C under $\sim$. We will denote by $[a]$ the equivalence class containing a.

Two distinct but equivalent character values are different representations of the same abstract character. The two representations should behave identically in all algorithms. For example, a don't-care character that matches all other characters is distinct from other characters and forms its own equivalence class. Its special matching properties are not part of the alphabet but a separate entity called a matching relation.

3.3 Conversions

The restriction to a single type applies to concrete alphabets but not abstract alphabets as multiple concrete alphabets can represent the same abstract alphabet. Conversions between concrete alphabets are the mechanism to deal with this.

Let $\mathcal{A}$ and $\mathcal{B}$ be two concrete alphabets. A *conversion* from $\mathcal{A}$ to $\mathcal{B}$ is a mapping $f\colon C^{\mathcal{A}} \to C^{\mathcal{B}}$ that is homomorphic w.r.t. $\sim$, i.e., $a \sim a' \Rightarrow f(a) \sim f(a')$ for all $a, a' \in \mathcal{A}$. Then, we can define $\widetilde{f}\colon \widetilde{\mathcal{A}} \to \widetilde{\mathcal{B}}$ by $\widetilde{f}([a]) = [f(a)]$. The following properties of $\widetilde{f}$ are of interest:

- $\widetilde{f}$ is an *embedding* if it is injective (one-to-one), i.e., $[a] \neq [a'] \Rightarrow \widetilde{f}([a]) \neq \widetilde{f}([a'])$.
- $\widetilde{f}$ is an *isomorphism* if it is a surjective embedding, i.e., an embedding satisfying $\widetilde{f}(\widetilde{\mathcal{A}}) = \widetilde{\mathcal{B}}$.

If there is an isomorphism $\widetilde{f}\colon \widetilde{\mathcal{A}} \to \widetilde{\mathcal{B}}$, we can say that $\mathcal{A}$ and $\mathcal{B}$ are two representations of the same abstract alphabet. Similarly, an embedding implies a subset relation.

The mapping $\widetilde{f}$ being an embedding or an isomorphism does not imply that the conversion f is injective or surjective. The following lemmas characterize embeddings and isomorphisms in terms of conversions.

Lemma 1. *$\widetilde{f}$ is an embedding iff $a \not\sim b \Rightarrow f(a) \not\sim f(b)$.*

Lemma 2. *$\widetilde{f}\colon \widetilde{\mathcal{A}} \to \widetilde{\mathcal{B}}$ is an isomorphism and $\widetilde{g}\colon \widetilde{\mathcal{B}} \to \widetilde{\mathcal{A}}$ is its inverse iff $\widetilde{f}$ and $\widetilde{g}$ are embeddings and $g(f(a)) \sim a$ for all $a \in \mathcal{A}$.*

Embedding conversions in particular play a central role in the library as we will see later. Isomorphic conversions come into play when inverse conversions are involved.

3.4 Ordered alphabets

A concrete *ordered* alphabet $\mathcal{A}$ is a quadruple $(\mathtt{T}, C, \sim, <)$, where $\mathtt{T}$, C and $\sim$ are as before and $<$ is a strict order on C satisfying: For all $a, b \in C$, exactly one of $a < b$, $a \sim b$ and $b < a$ is true. (We also define $\lesssim$ in the usual way.) The corresponding abstract ordered alphabet $\widetilde{\mathcal{A}}$ has an order $\preceq$ defined by $[a] \preceq [b]$ if $a \sim b$ or $a < b$.

A mapping $\widetilde{f}\colon \widetilde{\mathcal{A}} \to \widetilde{\mathcal{B}}$ is *order preserving* if it is homomorphic w.r.t. $\preceq$.

Lemma 3. *$\widetilde{f}$ is order-preserving iff f is homomorphic w.r.t. $\lesssim$.*
$\widetilde{f}$ is an order-preserving embedding iff f is homomorphic w.r.t. $<$.

Order preservation is a surprisingly subtle issue. There are common isomorphisms and embeddings that are not order-preserving. The standard conversion from `signed char` to `unsigned char` is an example. Also, order preservation is often not required even when order comparisons are involved. For example, the implementation of a character set using a balanced search tree requires a linear order but what the order is does not matter. A non-order-preserving conversion would not be a problem then. We will therefore not generally require conversions to be order preserving. However, when the *problem* definition involves an order, for example in the case of sorting, the conversions must be order preserving.

3.5 Integral Alphabets

Many algorithmic techniques work only or primarily on integral alphabets. These include using a character as an array index, computing fingerprints or hash values, radix sorting, etc. These techniques can be made available to a wide variety of alphabets through embeddings to proper integral alphabets.

A concrete alphabet $(\texttt{T}, C, \sim, <)$ is a *primary integral alphabet* if `T` is a built-in integral type (for example `char` or `int`), C is a range of the form $[0, \sigma)$, $\sim$ is the standard `operator==`, and $<$ is the standard `operator<`. Requiring the minimum to be zero simplifies many of the techniques mentioned above.

A concrete alphabet is a *secondary integral alphabet* if there is an embedding conversion f from it to a primary integral alphabet. An integer range with a minimum other than zero is a secondary integral alphabet, too.

Of additional interest is an isomorphic conversion *from* a primary integral alphabet. For example, random generation of characters can be accomplished using it.

4 Alphabet Traits

The character type `T` does not, in general, contain full information about the alphabet. Additional information in a form usable by algorithms is provided by *alphabet traits*. We will not describe the full design of alphabet traits but give a glimpse to their use with examples.

An alphabet traits is partly a C++ class and partly an object of that class. The class contains *static information* about the alphabet, i.e., information that is known at compile time and can be used for compile time optimization. An object of that class may contain additional *dynamic information*. For example, whether an alphabet is integral or not is always static information but the size of the integral range might be dynamic information.

4.1 Writing Generic Algorithms

The example in Section 2 shows the use of alphabet traits in writing generic algorithms at its simplest. Almost all details are hidden inside the metafunction `generate_set`, which is a part of the basic library infrastructure.

Obtaining more detailed information is demonstrated in the following example. Let `Alphabet` be an alphabet traits class and `A` an object of the class. If the alphabet is integral, we can obtain the conversion to a primary integral alphabet as follows:

```
get_char2int<Alphabet>::type char2int = make_char2int(A);
```

Then `char2int(ch)` performs the conversion for the character `ch`. Comparison functions, for example, are obtained similarly.

The above statement would not even compile for a non-integral alphabet. However, there are standard metaprogramming techniques for conditional compilation based on compile time predicates [1]. In this case, we can determine the integrality, at compile time, using the metafunction

```
is_integral<Alphabet>::value
```

As we saw in Section 2, alphabet traits is supplied as an argument to a function. To make things simpler for the caller of the algorithm, the argument should be optional. When no argument is supplied, the *default alphabet traits* for the character type is used instead. In the case of the `count_distinct` function, this is accomplished by providing the following second variant of the function.

```
template <typename Iterator>
int count_distinct(Iterator begin, Iterator end)  {
  typedef iterator_traits<Iterator>::value_type chartype;
  typedef default_alphabet<chartype>::type alphabet;
  return count_distinct(begin, end, alphabet());
}
```

4.2 Creating Alphabets

As mentioned, algorithms typically assume a default alphabet if no alphabet traits is provided by the user. If the default is not correct, the user needs to pass a correct one as an argument to the algorithm. The library will provide a number of alphabet traits for common situations. If none of these is satisfactory, there are metafunctions for creating custom alphabets.

The following example shows one way for creating a case-insensitive alphabet.

```
struct caseless_equal {
  bool operator() (char a, char b) {
    return tolower(a)==tolower(b);
  }
};
typedef construct_alphabet<char,
                  set_equivalence<caseless_equal> >::type
        caseless_alphabet;
```

Now a call such as `count_distinct(begin, end, caseless_alphabet())` would count upper and lower case letters as one.

The above alphabet is not ordered or integral as no order comparison or integral conversion is provided. Therefore, the set in `count_distinct` would be implemented as an unordered list. An order comparison and an integral conversion could be provided as additional arguments to the metafunction, but there is simpler way:

```
struct tolower_conversion {
  char operator() (char c) { return tolower(c); }
};
typedef embedded_alphabet<char, default_alphabet<char>::type,
                 tolower_conversion >::type
        caseless alphabet;
```

Here we create a new alphabet by embedding it to an existing alphabet. Many properties including ordering and integrality are automatically inherited. There is a similar metafunction `isomorphic_alphabet` that also takes the inverse conversion as an argument.

Integral alphabets are common and useful alphabets and there is a separate metafunction for creating alphabet traits for them. For example,

```
integral_alphabet<char, 10, 20>::type
```

creates an alphabet representing the range $[10, 20]$.

All the example alphabet traits here contain no dynamic information. Creating alphabet traits with dynamic information is more complicated and we ignore the details here.

5 Concluding Remarks

The purpose of this paper is to iniate the design of a string algorithms library based on the generic programming paradigm. We have addressed only one fundamental but limited aspect of the library, the alphabet. However, we believe that the design approach based on a careful analysis of concrete examples leading to a definition of the concept of an alphabet and the programming techniques developed for implementing the design provide a good start for the design of further aspects of the library.

The design of the sequence aspect has already been provided to an extent, thanks to the STL iterators and some further work building on them (`http://boost.org/libs/iterator/doc/`, `http://boost.org/doc/html/string_algo/design.html`, and `http://boost.org/libs/range/`). There are still issues remaining, though. For example, in some cases the alphabet and sequence aspects cannot be fully separated without a loss of efficiency [9].

Still more aspects are relevant to a string algorithms library. We have already mentioned one, match relation. Other issues arise, for example, from approximate string matching and other more complex stringology problems.

References

[1] D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond.* Addison–Wesley, 2004.

[2] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied.* Addison–Wesley, 2001.

[3] M. H. Austern. *Generic Programming and the STL.* Addison–Wesley, 1999.

[4] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, Oct. 1977.

[5] M. Crochemore and W. Rytter. *Jewels of Stringology.* World Scientific, 2002.

[6] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications.* Addison–Wesley, 2000.

[7] A. Czumaj, P. Ferragina, L. Gasieniec, S. Muthukrishnan, and J. L. Träff. The architecture of a software library for string processing. In *Proceedings of Workshop on Algorithm Engineering*, pages 294–305, 1997. Online proceedings at `http://www.dsi.unive.it/~wae97/proceedings/`.

[8] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the design of CGAL, the computational geometry algorithms library. *Software — Practice and Experience*, 30(11):1167–1202, 2000.

[9] K. Fredriksson. Faster string matching with super-alphabets. In *Proc. 9th International Symposium on String Processing and Information Retrieval (SPIRE)*, volume 2476 of *LNCS*, pages 44–57. Springer, 2002.

[10] K. Mehlhorn and S. Näher. *LEDA — A Platform for Combinatorial and Geometric Computing.* Cambridge University Press, 1999.

[11] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings.* Cambridge University Press, 2002.

[12] B. Smyth. *Computing Patterns in Strings.* Pearson Addison–Wesley, 2003.

[13] D. Vandevoorde and N. M. Josuttis. *C++ Templates: The Complete Guide.* Addison–Wesley, 2002.

A Full Example

Here is the full implementation of the `count_distinct` algorithm.

```
#include"glas/set.hpp"
#include<iterator>

// count the number of distinct characters in a string
// generic form with an alphabet as an argument
template< typename Iterator, typename Alphabet>
int
count_distinct(Iterator begin,
               Iterator end,
               Alphabet A)
{
  typedef typename glas::generate_set<Alphabet>::type charset;
  charset seen;
  for (Iterator i = begin; i != end; ++i) {
```

```
      seen.insert(*i);
  }
  return seen.size();
}

// specific form that uses default alphabet
template<typename Iterator>
int
count_distinct(Iterator begin, Iterator end)
{
  typedef typename std::iterator_traits<Iterator>::value_type
      char_type;
  typedef typename glas::default_alphabet<char_type>::type
      alphabet;
  return count_distinct(begin, end, alphabet());
}
```

Below is an program that uses the `count_distinct` function with a case insensitive alphabet.

```
#include "count.hpp"
#include "glas/alphabet_traits.hpp"
#include<string>
#include<iostream>

// case insensitive alphabet
struct tolower_conversion
{
  char operator() (char c) const { return tolower(c); }
};
struct caseless_alphabet
  : glas::embedded_alphabet<
       char,
       glas::default_alphabet<char>::type,
       tolower_conversion
    >::type
{};

int main()
{
  std::string str("ABRACAdabra");
  int cnt = count_distinct(str.begin(), str.end(),
                           caseless_alphabet());
  std::cout << cnt << " distinct characters in "
            << '"' << str << '"' << "\n";
  // prints: 5 distinct characters in "ABRACAdabra"
}
```

Flexible Music Retrieval in Sublinear Time

Kimmo Fredriksson[1]*, Veli Mäkinen[2]†, and Gonzalo Navarro[3]‡

[1] Dept. of Computer Science, University of Joensuu, Finland
e-mail: kfredrik@cs.joensuu.fi

[2] Technische Fakultät, Bielefeld Universität, Germany
e-mail: veli@cebitec.uni-bielefeld.de

[3] Dept. of Computer Science, University of Chile, Chile
e-mail: gnavarro@dcc.uchile.cl

Abstract. Music sequences can be treated as texts in order to perform music retrieval tasks on them. However, the text search problems that result from this modeling are unique to music retrieval. Up to date, several approaches derived from classical string matching have been proposed to cope with the new search problems, yet each problem had its own algorithms. In this paper we show that a technique recently developed for multipattern approximate string matching is flexible enough to be successfully extended to solve many different music retrieval problems, as well as combinations thereof not addressed before. We show that the resulting algorithms are close to optimal and much better than existing approaches in many practical cases.

Keywords: Music retrieval, approximate string matching, (δ, γ)-matching, transposition invariance.

1 Introduction

In this paper we are interested in music retrieval, and in particular, in a recent approach to it where musical scores are regarded as strings and string matching techniques can be used to solve music retrieval problems. In order to map the problem to string matching, the alphabet of the string could simply be the set of notes in the chromatic or diatonic notation, or the set of intervals that appear between notes (for example, pitches may be represented as MIDI numbers and pitch intervals as number of semitones). In both cases, we deal with *numeric* strings. Then, many music retrieval problems can be converted into string matching problems, that is, find the occurrences of a short string (called the *pattern*) in a longer string (called the *text*). This is usually not enough to fully solve all music retrieval problems, but it provides a useful and efficient filter to leave the most promising candidates for a

*Funded by the Academy of Finland, grant 202281.

†Funded by the Deutsche Forschungsgemeinschaft (BO 1910/1-3) within the Computer Science Action Program.

‡Partially funded by Millennium Nucleus Center for Web Research, Grant P04-067-F, Mideplan, Chile.

more profound and costly evaluation. There are also some problems where two long musical pieces are compared, which we do not address in this paper.

Exact string matching cannot be used to find occurrences of a particular melody, because a number of irrelevant distortions could exist between the melody sought and its version stored in the music database. To perform meaningful music retrieval one must resort to diverse forms of *approximate* matching, where a limited amount of *differences* of diverse kinds are permitted between the search pattern and its occurrence in the text. Different versions of the approximate string matching problem arise in different fields [24], yet those of music retrieval are unique of this area [11, 5, 28].

One approximate matching model of use in music retrieval is (δ, γ)-*matching*. In this model, two strings $a_1a_2 \ldots a_m$ and $b_1b_2 \ldots b_m$ of the same length m match if (i) the absolute differences between corresponding characters do not exceed δ, that is, $|a_i - b_i| \leq \delta$ for all $1 \leq i \leq m$ (or, alternatively, $\max_{1 \leq i \leq m} |a_i - b_i| \leq \delta$), and (ii) the sum of those absolute differences does not exceed γ, that is, $\sum_{1 \leq i \leq m} |a_i - b_i| \leq \gamma$. This model accounts for small differences that may arise between two versions of the same melody, setting a limit for the individual absolute differences, as well as a global limit to the overall differences. Searching for pattern p under (δ, γ)-matching consists of finding all the text positions where a text substring that (δ, γ)-matches p appears. Less popular subproblems are δ-matching and γ-matching, which only enforce one of the two conditions.

A second relevant approximate matching model is the *longest common subsequence (LCS)* and its dual *indel distance*. The former, $LCS(a, b)$, is the maximum length of a string that is subsequence both of a and b, that is, $LCS(a, b) = \max\{|s|,\ s \sqsubseteq a,\ s \sqsubseteq b\}$. A string $s = s_1 s_2 \ldots s_r$ is a *subsequence* of string $a_1a_2 \ldots a_m$, $s \sqsubseteq a$, if s can be obtained by removing zero or more characters from a, that is, $s = a_{i_1}a_{i_2} \ldots a_{i_r}$ for $1 \leq i_1 < i_2 < \ldots < i_r \leq m$. The LCS has been largely used in computational biology to model biological similarity, and it is also relevant to identify musical passages that are similar except for a few extra or missing notes. This is especially relevant because music contains various kind of "decorations", such as grace notes and ornamentations, that are not essential for matching. The indel distance $id(a, b)$ between strings a and b is the number of characters one has to add or remove to a and b to make them equal, $id(a, b) = |a| + |b| - 2 \cdot LCS(a, b)$. Searching for pattern p under indel distance with tolerance k consists of finding all the text positions where a string p' appears so that $id(p, p') \leq k$. Other variants of indel distance, which are less popular in music retrieval, are Levenshtein or edit distance (where substitutions of characters are also permitted) and episode matching (where only insertions in the pattern are permitted).

Finally, a third similarity concept of relevance in music retrieval is *transposition invariance*. Two strings $a = a_1a_2 \ldots a_m$ and $b = b_1b_2 \ldots b_m$ are one the transposed version of the other if there is a constant t such that $a+t = (a_1+t)(a_2+t) \ldots (a_m+t) = b$. Transposition invariance is very relevant because Western people tend to listen to music analytically, by observing the intervals between consecutive pitch values rather than the actual pitch values themselves. As a result, a melody performed in two distinct pitch levels is perceived as equal regardless of whether it is performed in a lower or higher level of pitches.

As a string matching problem, dealing with transposition invariance is trivial because it suffices to represent text and pattern as differences between consecutive

notes and then apply exact string matching. However, the above problems in most cases of interest appear in combined form. In particular, transposition invariance is usually combined with longest common subsequence. The longest common transposition invariant subsequence between two strings a and b, $LCTS(a, b)$, permits transposing a or b as necessary to find the longest common subsequence among them, $LCTS(a, b) = \max_{t \in \mathbb{Z}} LCS(a + t, b)$.

In recent years, there has been much activity around developing specific string matching techniques to solve diverse music retrieval problems, mostly consisting of combinations of those outlined above. Several theoretical and practical results of interest have been achieved. We cover these in the next section.

Our contribution in this paper is to show that a particular approach recently developed for multiple approximate string matching [17] is flexible enough to be successfully adapted to solve most of the combinations of problems sketched above. Basically the same search technique, coupled with slightly different pattern preprocessings, yield algorithms that solve each combination. We also characterize those combinations that cannot be addressed by our approach. In theoretical terms, we show that the resulting algorithms are sublinear (that is, they do not inspect all text characters) and can be argued to be close to optimal. Yet, the most important aspect is the practical side, where we show that our technique largely outperforms all the existing ones in most cases of interest.

2 Related Work

In which follows, we assume that a long text $T = t_1 t_2 \ldots t_n$ is searched for a comparatively short pattern $p = p_1 p_2 \ldots p_m$. Both are sequences over alphabet Σ, a finite contiguous subset of $\mathbb{Z}$, of size σ.

2.1 (δ, γ)-Matching

Several recent algorithms exist to solve this problem. These can be classified as follows:

Bit-parallel: The idea is to take advantage of the intrinsic parallelism of the bit operations inside a computer word of w bits [27], so as to pack several values in a single word and manage to update them all in one step [6, 7, 13]. The best complexity achieved [13] is $O(n\ m \log(\gamma)/w)$ in the worst case and $O(n)$ on average.

Occurrence heuristics: Inspired by Boyer-Moore techniques [4], they skip some text characters according to the position of some characters in the pattern [6, 12]. In general, only δ is used to skip characters, while the γ-condition is used to verify candidates. This makes these algorithms weak for large δ and small γ.

Substring heuristics: Based on suffix automata [15], these algorithms skip text characters according to the position of some pattern substrings [12, 13]. In the second article, they use bit-parallelism to filter the text using both δ and γ, unlike previous approaches. This is shown to be the approach examining the least number of text characters.

FFT-related: It is possible to solve the δ-matching and (δ, γ)-matching problems in $O(\delta n \log m)$ time, and γ-matching problem in $O(n\sqrt{m \log m})$ time [8] using Fast Fourier Transform (FFT) based techniques. The $O(n\gamma \log \gamma)$ time algorithm in [2] is faster for small γ. This algorithm is based on bounded divide-and-conquer and non-boolean convolutions. This technique can be also used to solve the δ-matching problem in $O(n \log m\sqrt{\delta})$ time. Other FFT based $o(mn)$ solutions exist for related problems, see e.g. [9] and especially related to δ-matching [1, 10]. Matching under γ-restriction is possible in $O(mn/\log_\sigma n)$ time [22] without using FFT (but using the Four-Russians trick).

In practice, the best current algorithms for (δ, γ)-matching are those in [13], as demonstrated by the experiments in [12, 13]. In [13] they present a plain bit-parallel and a substring heuristic. The first is shown to be the best in most cases, but for short patterns and small δ and γ, the character-skipping technique is better.

The FFT based techniques, although elegant, have considerably large overheads to make them practical. Our preliminary tests show that they only become faster than the naive algorithm on very long patterns. Searching for long patterns is not typical in music retrieval. The solution based on the Four-Russians trick is only practical for small alphabets, much smaller than what is required for music retrieval.

2.2 Transposition Invariant LCS and Indel Distance

Plain (non-transposed) LCS among strings p and T can be computed in $O(mn)$ time using dynamic programming [18]. In general, any LCTS algorithm can be adapted to text searching with indel distance. The LCTS problem was first stated in [21], where $O(\sigma mn)$ time was obtained by trying out all the $2\sigma + 1$ possible transpositions one by one. Further solutions to the problem can be classified as follows.

Brute-force: The idea is to pick any LCS algorithm and try it for all the $2\sigma + 1$ possible transpositions. Apart from the original proposal [21], several others have been attempted considering different practical LCS algorithms based on bit-parallelism [14, 19]. The best complexity achieved is $O(\sigma mn/w)$.

Sparse dynamic programming: An evolution over the above scheme is to notice that the $LCS(a + t, b)$ problem for each transposition t has only a few character matches between a and b, mn in total. Those *sparse* problems are best handled by sparse dynamic programming algorithms. This idea lead to several solutions [23, 26, 16]. The best complexity achieved is $O(mn \log\log \min(m, \sigma))$, yet a version with complexity $O(mn \log \sigma / \log w)$ is shown to be better in practice.

Branch and bound: In this case the idea is to search for the best possible transposition t by a backtracking method, recursively dividing the space of $2\sigma + 1$ transpositions into ranges until finding the best one [20]. This yields a best-case complexity of $O((mn + \log\log \sigma) \log \sigma)$, and the method works well in practice. Yet, it cannot be extended to searching with indel distance.

Experiments in [20, 19, 16] demonstrate that the $O(mn \log \sigma / \log w)$ algorithm in [16] is the fastest in practice. This method can be adapted to searching with indel distance.

3 Optimal Multiple Approximate String Matching

In [17], new algorithms for single and multiple approximate string matching were presented. Those algorithms were not only optimal on average, but also very efficient in practice, even in the more competitive area of single approximate string matching. It was shown that, to search for the occurrences of r patterns of length m in a text of length n, all them uniformly distributed over an alphabet of size σ, the algorithm required $O(n(k + \log_\sigma(rm))/m)$ time on average. Here k is the maximum number of missing, extra, or substituted characters permitted to match a pattern against a text string (searching under edit distance). This average complexity is optimal [29, 25].

We first explain how to search for a single pattern p. We choose a *block length* ℓ, and compute $med(b, p)$ for every possible block $b \in \Sigma^\ell$ (that is, every possible ℓ-*gram*). Here, $med(b, p)$ is the minimum edit distance between b and a substring of p,

$$med(b, p) \quad = \quad \min\{ed(b, p'),\ \exists x, y,\ p = xp'y\},$$

being $ed(b, p')$ the edit distance between b and p'.

Now, the text $T = t_1 t_2 \ldots t_n$ is scanned as follows. Since the minimum length of an occurrence of $p = p_1 p_2 \ldots p_m$ in T with edit distance at most k has length at least $m-k$ (when k deletions occur on p), we slide a window of length $m-k$ along the text. For each window tried, $t_{i+1} t_{i+2} \ldots t_{i+m-k}$, we read its ℓ-grams right to left. That is, we read at most $\lfloor (m-k)/\ell \rfloor$ ℓ-grams b_1, b_2, and so on, so that $b_1 = t_{i+m-k-\ell+1} \ldots t_{i+m-k}$ is the rightmost, $b_2 = t_{i+m-k-2\ell+1} \ldots t_{i+m-k-\ell}$ precedes b_1, etc. The invariant is that any occurrence of p starting at positions $\leq i$ has already been reported.

For each such ℓ-gram $b_j = t_{i+m-k-j\ell+1} \ldots t_{i+m-k-j\ell+\ell}$, we find $med(b_j, p)$ in the precomputed table. If, after reading b_j, we have $med(b_1, p) + med(b_2, p) + \ldots + med(b_j, p) > k$, then no possible occurrence of p can contain the text $b_j b_{j-1} \ldots b_2 b_1$, thus the window is slid forward to start at the second character of b_j, that is, we set $i \leftarrow i + m - k - j\ell + 1$ (as the new window will start at $i + 1$).

If, on the other hand, all the ℓ-grams of the window are scanned and yet the window cannot be shifted, it must be verified for a real occurrence. At this point, we must check if there is an occurrence p' of p starting at text position $i + 1$. Since the maximum length of an occurrence is $m + k$ (where k insertions occur into p), any potential p' mush finish between positions $i + m - k$ and $i + m + k$. So we compute

$$led(p, i) \quad = \quad \min\{ed(p, t_{i+1} \ldots t_{i+m-k+d}),\ 0 \leq d \leq 2k\},$$

which can be done in $O(m^2)$ time by computing $ed(\ )$ incrementally in d. If $led(p, i) \leq k$, we report $i + 1$ as the starting position of an occurrence. Finally, we advance the window by one position, $i \leftarrow i + 1$.

We show now that the way we shift the window is safe, that is, no occurrence can start at positions $i + 1$ to $i + m - k - j\ell + 1$. Any such occurrence, of length at least $m - k$, must contain the sequence of ℓ-grams $b_j \ldots b_1$. Let $p' = xb_j \ldots b_1 y$ be such an occurrence. This is a split of p' into $j + 2$ pieces. The main point is that the edit distance is *decomposable*: For any strings p and p', given any split $p' = p'_1 \ldots p'_{j+2}$, there is a split $p = p_1 \ldots p_{j+2}$ such that $ed(p', p) = ed(p'_1, p_1) + \ldots + ed(p'_{j+2}, p_{j+2})$. But each such $ed(p'_s, p_s) \geq med(p'_s, p) \geq 0$, by definition of $med(\ )$.

Hence, in our particular case, $ed(p', p) \geq med(b_j, p) + \ldots + med(b_1, p)$. Thus if the latter exceeds k, there can be no occurrence of p containing $b_j \ldots b_1$.

The extension of the algorithm for multiple patterns is trivial. We only have to change the preprocessing so that p is now a set of patterns $p = \{p^1 \dots p^r\}$ and now $med(b, p) = \min_{1 \le i \le r} med(b, p^i)$. So $med(b, p)$ is a lower bound to the cost of matching b anywhere inside any pattern of the set.

By appropriately choosing $\ell = \Theta(\log_\sigma(rm))$, we obtain the promised complexity.

3.1 Extensions

Several other improvements are studied in [17]. We briefly review some that are used in our experiments. For more details see [17].

On the windows that have to be verified, we could simply run the verification for every pattern, one by one. A more sophisticated choice is *hierarchical verification* [3]. We form a tree whose nodes have the form $[i, j]$ and represent the group of patterns $p^i \dots p^j$. The root is $[1, r]$, and the leaves have the form $[i, i]$. Every internal node $[i, j]$ has two children $[i, \lfloor (i + j)/2 \rfloor]$ and $[\lfloor (i + j)/2 \rfloor + 1, j]$.

The preprocessing is done first for the leaves, as in the single pattern case, that is, we compute a table for $med(b, p^i)$. The internal nodes contain tables for $\min_{i \le h \le j} med(b, p^h)$, computed as minimizing over the two tables of the subtrees. In the filtering phase, we first use the table for the root, corresponding to the full set of patterns, and if the current window has to be verified with respect to a node in the hierarchy, we rescan the window considering the two children of the current node. It is possible that the window can be discarded for both children, for one, or for none. We recursively repeat the process for every child that does not permit discarding the window. If we process a leaf node and still have to verify the window, then we run the verification algorithm for the corresponding single pattern.

The second improvement is to have *bit-parallel counters*. In this case we reserve only $O(\log_2 k)$ bits to accumulate the differences $med(b_j, p)$. This means that if we have a computer word of w bits, we can process $O(w/\log_2 k)$ patterns in parallel. This technique can also be used with the hierarchical verification, to increase the arity of the tree to $O(w/\log_2 k)$.

The third improvement is to use *ordered ℓ-grams*, where each b_j is permitted to match only in the area of p where it could be aligned in an occurrence starting at $i + 1$. In an approximate occurrence of $b_j \dots b_1$ inside the pattern, b_i cannot be closer than $(i - 1)\ell$ positions to the end of the pattern. Therefore, we compute tables for $med^j(b, p)$, $1 \le j \le \lfloor (m - k)/\ell \rfloor$, where $med^j(b, p) = \min\{ed(b, p'),\ \exists x, y,\ |y| \ge (j - 1)\ell,\ p = xp'y\}$. This allows us to discard a window whenever $med^1(b_1, p) + med^2(b_2, p) + \dots + med^j(b_j, p) > k$. This reduces verifications but increases preprocessing time and space.

Finally, it is possible to improve the preprocessing time by using a trie of all the possible ℓ-grams to reuse preprocessing work. All the improvements can be combined into a single algorithm.

4 Adapting to Music Retrieval

The method above was designed for multiple string matching under edit distance. Yet its main idea is much more general and can be used to solve many other problems. In this section we demonstrate that the idea solves most of the music retrieval problems we have focused on in this paper. We note that this gives immediately a solution to the multipattern version of the same problems.

4.1 Transposition Invariant Indel Distance

Let us start with searching with transposition invariant indel distance. For each ℓ-gram $b \in \Sigma^\ell$, we compute

$$mtid(b,p) \quad = \quad \min\{id(b+t,p'),\ \exists x,y,\ p=xp'y,\ \ -\sigma \le t \le \sigma\}.$$

This is the minimum transposition invariant indel distance to match b anywhere inside p. The same algorithm of the previous section is used, and the same argument shows that we cannot discard a window that starts an occurrence of p in T. Indel distance is decomposable just like edit distance, that is, for any split $p' = p'_1 \dots p'_{j+2}$, there is a split $p = p_1 \dots p_{j+2}$ such that $id(p',p) = id(p'_1,p_1) + \dots + ed(p'_{j+2},p_{j+2})$. Assume p matches t the current window $xb_j \dots b_1 y$ starting at position $i+1$. That is, there exists a transposition t such that $id(p',p) \le k$, $p' = (x+t)(b_j+t)\dots(b_1+t)(y+t)$. Now, $id(p',p) \ge id(b_j+t,p_2) + \dots id(b_1+t,p_{j+1}) \ge mtid(b_j,p) + \dots mtid(b_1,p)$. Thus if the latter exceeds k we can safely shift the window.

When a window starting at position $i+1$ cannot be shifted, we simply compute $LCTS(p, t_{i+1} \dots t_{i+m-k+d})$ for any $0 \le d \le 2k$, and report position $i+1$ if $LCTS(p, t_{i+1} \dots t_{i+m-k+d}) \ge (m+m-k+d-k)/2 = m-k+d/2$ for some d, as this is equivalent to $id(p, t_{i+1} \dots t_{i+m-k+d}) \le k$ for some transposition t.

Fig. 1 shows simplified pseudocode.

4.2 (δ,γ)-Matching

Alternatively, we can search for (δ,γ)-matches of p in T. In this case the window is of length m, as occurrences are all of that length. For each ℓ-gram $b \in \Sigma^\ell$, we compute

$$mdg(b,p) \quad = \quad \min\{\gamma',\ \exists x,y,\ p=xp'y,\ b\ (\delta,\gamma')\text{-matches } p'\}.$$

This is the minimum total number of absolute differences obtained by b inside p, where we restrict those positions to δ-match as well. The same algorithm of the previous section is used with this preprocessing (and the threshold is γ instead of k).

Being γ-matching a cumulative measure, the sum of $mdg(b_j,p)$ values is a lower bound to the γ needed to match the window inside p. Consider window $p' = t_{i+1} \dots t_{i+m} = xb_j \dots b_1$. Assume p' (δ,γ)-matches p. Then, by definition of (δ,γ)-matching, b_1 (δ,γ_1)-matches $p_{m-\ell+1} \dots p_m$, and so on until b_j, which (δ,γ_j)-matches $p_{m-j\ell+1} \dots p_{m-j\ell+\ell}$, so that $\gamma_1 + \dots + \gamma_j \le \gamma$. As each b_s (δ,γ_s)-matches $p_{m-s\ell+1} \dots p_{m-s\ell+\ell}$, it holds $mdg(b_s,p) \le \gamma_s$, and $mdg(b_j,p) + \dots + mdg(b_1,p) \le k$.

When a window $t_{i+1} \dots t_{i+m}$ cannot be shifted, we check whether p (δ,γ)-matches the window in time $O(m)$, and report position $i+1$ if this is the case.

Search ( )
1. $D \leftarrow$ **Preprocess** ( )
2. $i \leftarrow 0$
3. **While** $i \leq n - (m - k)$ **Do**
4. $\quad pos \leftarrow$ **Shift** $(i,\ D)$
5. $\quad$ **If** $pos = i$
6. $\quad\quad$ Verify area $t_{i+1} \ldots t_{i+m+k}$
7. $\quad\quad pos \leftarrow pos + 1$
8. $\quad i \leftarrow pos$

Shift $(i,\ D)$
1. $M \leftarrow 0$
2. $c \leftarrow m - k$
3. **While** $c \geq \ell$ **Do**
4. $\quad c \leftarrow c - \ell$
5. $\quad M \leftarrow M + D[t_{i+c+1} \ldots t_{i+c+\ell}]$
6. $\quad$ **If** $M > k$ **Return** $i + c + 1$
7. **Return** i

Preprocess ( )
1. $\ell \leftarrow \Theta(\log_\sigma m)$
2. **For** $b \in \Sigma^\ell$ **Do** $D[b] \leftarrow mtid(b, p)$
3. **Return** D

Figure 1: Simple description of the algorithm. The main variables are global for all the algorithms. The code corresponds to transposition invariant indel.

The pseudocode of Fig. 1 can be easily adapted to this model. One needs only to replace $mtid()$ with $mdg()$, k with γ, and adjust the window size from $m - k$ to m, and verification area from $t_{i+1} \ldots t_{i+m+k}$ to $t_{i+1} \ldots t_m$.

4.3 Feasible and Unfeasible Combinations

We can also combine transposition invariant indel distance with δ-matching. In this case we count indels, but two characters match whenever they do not differ by more than δ units. This is easily handled by modifying $mtid(b, p)$ formula so that $id(b+t, p')$ considers matches in the more relaxed way. Transposition invariance can also be combined with (δ, γ)-matching, by using $mtdg(b, p)$ instead of $mdg(b, p)$, so that

$$mtdg(b, p) \quad = \quad \min\{\gamma',\ \exists x, y,\ p = xp'y,\ b + t\ (\delta, \gamma')\text{-matches } p',\ -\sigma \leq t \leq \sigma\}.$$

We cannot directly combine transposition invariant indel distance with (δ, γ)-matching. The reason is that we do not have here a single value to minimize, such as the number of indels or γ, but both of them at the same time. It was possible to combine transposition invariant indel distance with δ-matching because the latter is not a parameter to optimize but a condition for matching. Likewise, it was possible to combine γ-matching with δ-matching to obtain (δ, γ)-matching. Yet, if we want to combine indel distance (even without transposition invariance) with γ-matching, the problem is that each pair (b, p') produces some number of indels and some γ, so different pairs will yield the minimal of each and it is not clear which to choose.

Of course we can count indels and γ separately in different tables (each achieved by a different pair). This is equivalent to filtering each window with k and with γ separately, and verifying those that pass both filters. Yet, this is not the same as a combined filter, but it could be practical.

4.4 Complexity Considerations

We are not able to analyze our algorithms, but we can give some clues about their average case performance. As we have described it, our algorithm for transposition invariant indel distance is equivalent to multipattern search with indel distance for the set $p^1 = p-\sigma, p^2 = p-\sigma+1, \ldots, p^{2\sigma+1} = p+\sigma$. Since $id(a,b) \geq ed(a,b)$ for any strings a and b, we can use the analysis of [17] on edit distance for indel distance and the result is pessimistic (yet tight). According to that analysis, searching for $r = 2\sigma + 1$ random patterns in random text yields average complexity $O(n(k + \log_\sigma(rm))/m) = O(n(k + \log_\sigma m)/m)$. This value is optimal even for one pattern [29], and it would show that our algorithm is optimal too.

Yet, the problem is that our $2\sigma + 1$ patterns are *not* random, but are all the transpositions of a random pattern. For example, if $\ell = 1$, then our $2\sigma + 1$ patterns necessarily match any string of length 1, whereas the same number of random patterns do not. Thus our analysis is optimistic and therefore not conclusive. Yet, we conjecture that the result of the analysis is valid.

In case δ-matching is permitted together with transposition invariance indel distance in the model, then the probability of matching is not $1/\sigma$ but $O(\delta/\sigma)$, and therefore the base of the logarithm is not σ but $O(\sigma/\delta)$. Redoing the analysis we get $O(n(k+\log_{\sigma/\delta}(\delta m))/m)$. With δ-matching alone (no transposition invariance) we get $O(n \log_{\sigma/\delta} m/m)$, and with δ-matching with transposition invariance (without indels) we get $O(n \log_{\sigma/\delta}(\delta m)/m)$. We are not able to account for the analytical effect of a γ-restriction in these analyses, but of course they can only improve.

In the worst case the filtering algorithm for each model takes $O(mn)$ time without the hierarchical verification, and $O(mn\sigma)$ with hierarchical verification (for the transposition invariant models). There is also a linear time variant of the filtering algorithm that runs in $O(n)$ time in the best and worst cases, see [17]. However, in the worst case the verification time dominates. For transposition invariant indel distance the worst case verification time is $O(nm\sigma/w)$. For (δ,γ)-matching the worst cases are $O(nm)$ without transpositions and $O(nm\sigma)$ with transpositions. We note that these can be improved by using the more efficient worst case algorithms available in the literature.

The preprocessing time is $O(m\sigma^{\ell+1}/w)$ for transposition invariant indels, $O(m\sigma^\ell)$ for (δ,γ)-matching, and $O(m\sigma^{\ell+1})$ for transposition invariant (δ,γ)-matching. With ordered ℓ-grams the preprocessing cost for indels increases to $O(m\sigma^{\ell+1})$. For the other models the costs remain the same. The space requirement is $O(\sigma^\ell)$ and $O(\sigma^\ell m/\ell)$ for the basic algorithm and for the ordered ℓ-grams, respectively. These have to be multiplied by $O(\sigma)$ if hierarchical verification is used. All the bounds are polynomial in m (as $\ell = \Theta(\log_\sigma m)$).

5 Experimental Results

We have implemented the algorithms in C, compiled using `icc 8.0` with full optimizations. The experiments were run in a 2GHz Pentium 4, with 512MB RAM, running Linux 2.4.18. The computer word length is $w = 32$ bits.

For the text we used a concatenation of 7543 music pieces, whose total length is 1828089 bytes. The file was obtained by extracting the pitch values from MIDI

files. The pitch values are in the range $[0 \ldots 127]$. A set of 100 patterns were randomly extracted from the text. Each pattern was then searched for separately, and we report the average search times. We measured user times. We have separated the preprocessing and search times, which makes it easier to compare the search performance. Our preprocessing cost is considerably high, but this is amortized by large music collections that arise in practical applications.

5.1 Implementation

Several variants of the optimal multipattern algorithm were considered in [17]. For (δ, γ)-matching without transpositions, we used the basic single pattern algorithm. As the transpositions were implemented as multipattern search, we used bit-parallel counters and hierarchical verification in these cases, which give a considerable speedup. For indels, we used the IndelMYE algorithm [19] for the final verifications. We ran each experiment with and without ordered ℓ-grams. The former is an order of magnitude faster in many cases, but it has higher preprocessing cost, justified only for large texts.

For all experiments we used $\ell = 2$. Due to the considerably large alphabet size, larger ℓ values were not practical. On the other hand, $\ell = 1$ gives in general poor results, especially combined with transpositions (but note that with bit-parallel counters even 1-grams are not guaranteed to match always, as different transposition ranges are mapped to different counters).

As the alphabet size was large (128), but most of the values occur in the middle of the range, we mapped the alphabet into the range $0 \ldots 63$. That is, values $32 \ldots 95$ were mapped to $0 \ldots 63$, values $0 \ldots 31$ to 0, and values $96 \ldots 127$ to 95. This mapping allows us to use the original δ values. Verification was done using the original alphabet. This improves the preprocessing times, without worsening the search times.

We note that other alphabet mappings may make sense. In particular, for music applications, it might be acceptable to make the alphabet octave-independent, so that the same notes in different octaves are mapped to the same value.

5.2 Preprocessing Time

Table 1 gives the preprocessing times. For $mtid()$ and $mtdg()$ we have considered hierarchical verification because it gave consistently better results, so the preprocessing timings include all the hierarchy construction. Using ordered ℓ-grams increases the preprocessing cost, but improves the search performance.

$mtid()$, $m = 32$	$mdg()$, $m = 8$	$mdg()$, $m = 64$	$mtdg()$, $m = 32$
0.0699 / 0.2680	0.0048 / 0.0052	0.0067 / 0.0092	0.0936 / 0.5177

Table 1: Preprocessing times in seconds for $\ell = 2$. The second timings are for ordered ℓ-grams.

5.3 Transposition Invariant Indel Distance

We compared our approach against the LCTS algorithm [16], whose running time is $O(mn \log \sigma / \log w)$. Although the algorithm solves the dual problem, it could be adapted to searching with indel distance as well. We also compared against the bit-parallel dynamic programming algorithm IndelMYE [19], whose running time for a single transposition is $O(mn/w)$. We superimposed [3] all the transpositioned patterns and used hierarchical verification, in the same manner as in [17] with BPM algorithm. This works very well in practice, although the worst case complexity is still $O(\sigma mn/w)$. Fig. 2 shows the results for $m = 8 \ldots 64$ and $k = 1 \ldots 5$. Our algorithm is by far the fastest for small k/m. LCTS is competitive only for very large k/m, while IndelMYE is the best choice for moderate k/m. Our algorithm clearly improves with ordered ℓ-grams, at the cost of higher preprocessing effort and memory requirements.

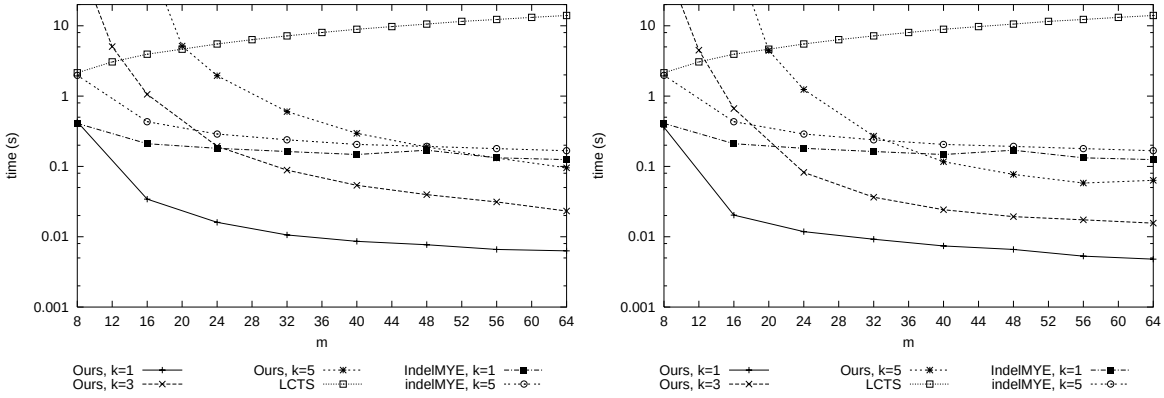


Figure 2: Left: Search time in seconds for transposition invariant indel/LCS for $m = 8...64$. Right: The same with ordered ℓ-grams.

Fig. 3 shows the results for $m = 32$, $k = 1 \ldots 6$ and $\delta = 0 \ldots 2$. The LCTS algorithm cannot be applied for this setting. Being bit-parallel algorithm, IndelMYE can be easily adapted to this case by using classes of characters to implement δ. In this case we are again competitive against IndelMYE for small k/m, but only for very small δ. Ordered ℓ-grams boost the search considerably.

5.4 (δ, γ)-Matching

For (δ, γ)-matching we compared against the bit-parallel Forward matching algorithm (Fwd) of [13]. Fig. 4 shows the results for $m = 8 \ldots 64$, $\delta = 1 \ldots 3$ and $\gamma = m\delta/2$. Our algorithm is much more sensitive to increasing δ than Fwd, but for small δ values we are an order of magnitude faster. Using ordered ℓ-grams makes our algorithm more tolerant for increasing γ (but note that γ/m is constant here).

In [13] they give also bit-parallel backward matching algorithm, that is able to skip some text characters. The implementation restricts the pattern lengths to be at most $\Theta(w/\log_2(\gamma))$. This means that in this experiment this algorithm is applicable only for the case $m = 8$, $\delta = 1$, and $\gamma = 8 * 1/2 = 4$. The algorithm takes 0.0063s average time, in this case, and marginally beats our algorithm (0.0065s)

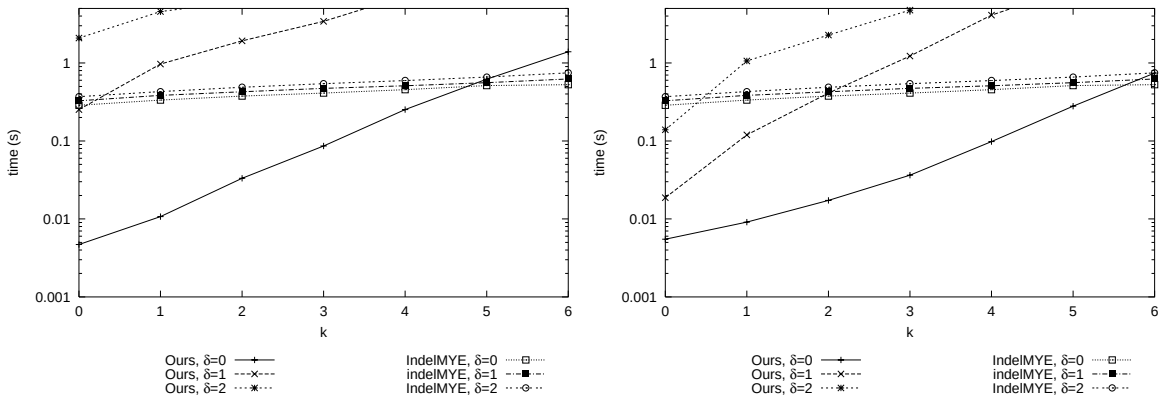


Figure 3: Left: Search times in seconds for transposition invariant indel for $\delta = 1...3$, and $m = 32$. Right: The same with ordered ℓ-grams.

Timings for $m = 32$, $\delta = 1 \ldots 3$, and $\gamma = 4 \ldots 40$ are shown in Fig. 5. (Note that for $\delta = 1$ there is no point for using $\gamma > m$.) Again, Fwd becomes eventually faster for large δ and γ, while our algorithm dominates for small parameter values. Fig. 6 repeats the experiment for transposition invariant (δ, γ)-matching. Note that no competitors exist in this case, although transposition superimposition and hierarchical verification could be applied for some of the existing (δ, γ) matching algorithms. However, observe that our transposition invariant algorithm is faster than Fwd algorithm (without transpositions) for small δ and γ.

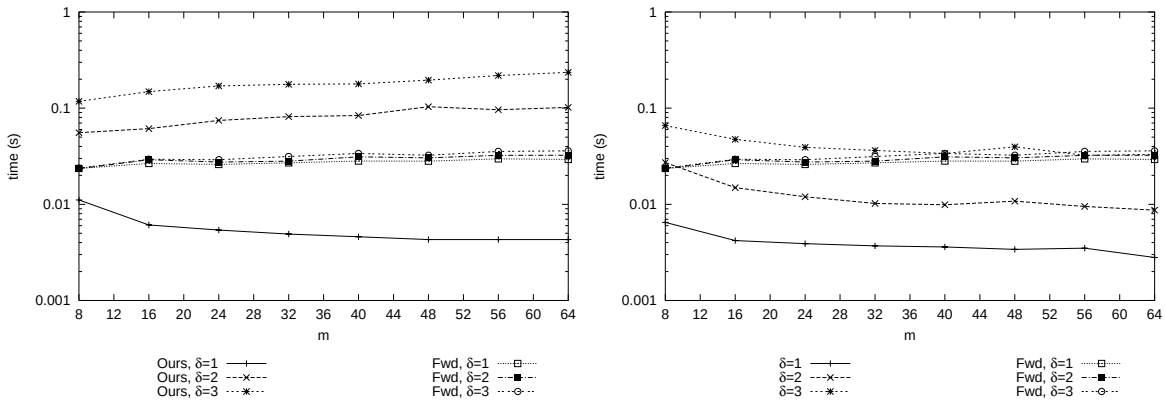


Figure 4: Left: Search times in seconds for (δ, γ)-matching for $m = 8 \ldots 64$ and $\delta = 1 \ldots 3$. For each data point $\gamma = m\delta/2$. Right: The same with ordered ℓ-grams.

5.5 Comparison

We have separated the preprocessing and searching times in presenting the experimental results. This may seem unfair against the competing algorithms, and so it is for short texts. To show that our algorithms *are* competitive, Table 2 gives estimates for the minimum file sizes required to beat the competing approaches for various problem instances. These limits are quite modest, and for smaller parameter values even shorter files are sufficient.

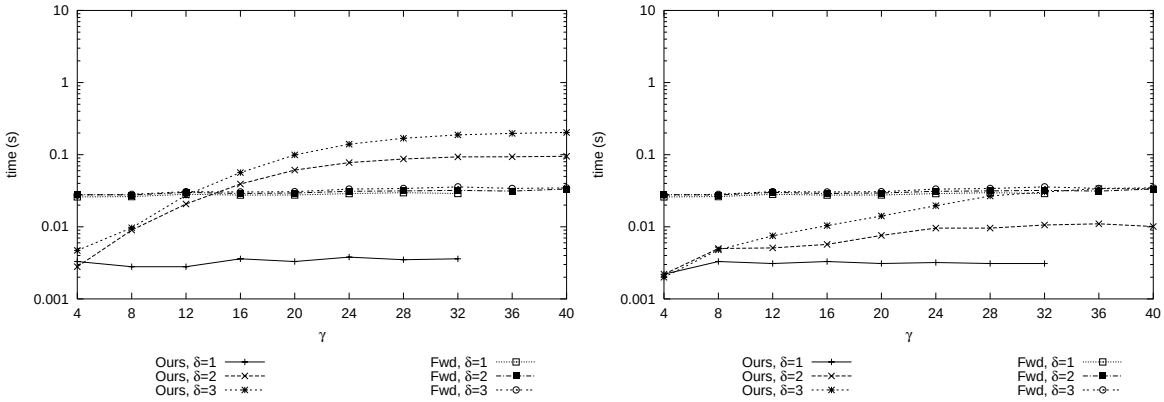


Figure 5: Left: Search times in seconds for (δ, γ)-matching for $m = 32$, $\delta = 1 \ldots 3$, and $\gamma = 4 \ldots 40$. Right: The same with ordered ℓ-grams.

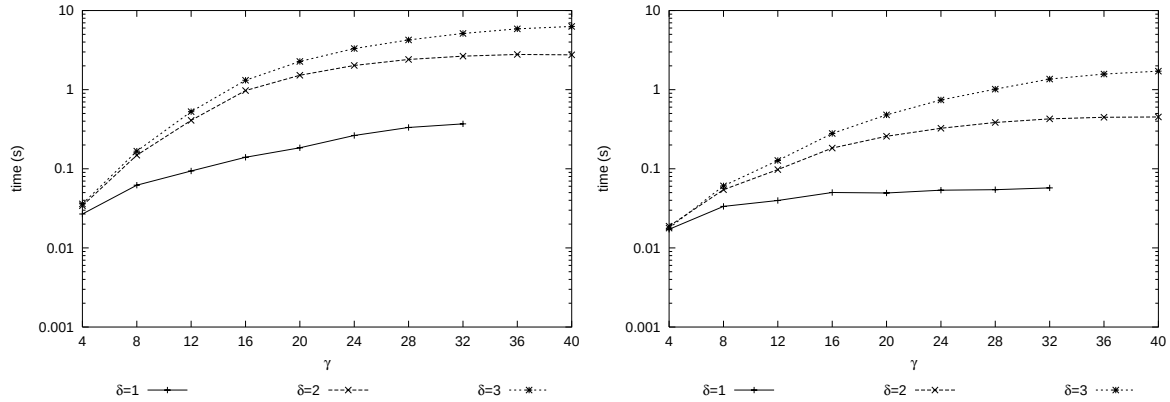


Figure 6: Left: Search times in seconds for (δ, γ)-matching with transpositions for $m = 32$, $\delta = 1 \ldots 3$, and $\gamma = 4 \ldots 40$. Right: The same with ordered ℓ-grams.

6 Conclusions

We have presented new filtering algorithms for music retrieval. Our algorithms are very efficient in practice, and are conjectured to be optimal on average. The experiments show that for small to moderate error thresholds our algorithms are substantially faster than previous approaches for all but very short texts. These are the parameter values that are most interesting in most music retrieval applications.

The algorithms are extremely flexible. We can solve many different problem variants essentially without any modifications to the search algorithms, only preprocessing changes according to the search model. In particular, we are able to solve some variants where no competing algorithms currently exist. These are transposition invariant indel with $\delta > 0$, and transposition invariant (δ, γ)-matching. Moreover, our algorithms can be used for multipattern search as well.

Indels	
$k = 4, \delta = 0$	$k = 1, \delta = 1$
> 0.61 Mb	> 1.77 Mb

(δ, γ)-matching		
$(1, \infty)$	$(2, \infty)$	$(3, 24)$
> 0.46 Mb	> 0.71 Mb	> 1.52 Mb

Table 2: Examples of music file sizes where we begin to win for a few settings. The first row shows the parameter values, and the second row gives an estimate of the minimum file size where our algorithm wins its competitor. For smaller parameters shorter files would suffice. The estimates are for $m = 32$.

References

[1] A. Amir and M. Farach. Efficient 2-dimensional approximate matching of half-rectangular figures. *Information and Computation*, 118(1):1–11, 1995.

[2] A. Amir, O. Lipsky, E. Porat, and J. Umanski. Approximate matching in the L_1 metric. In *Proc. CPM'05*, LNCS v. 3537, pages 91–103, 2005.

[3] R. Baeza-Yates and G. Navarro. New and faster filters for multiple approximate string matching. *Random Structures and Algorithms*, 20:23–49, 2002.

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

[5] E. Cambouropoulos, T. Crawford, and C. Iliopoulos. Pattern processing in melodic sequences: Challenges, caveats and prospects. In *Proc. AISB'99*, pages 42–47, 1999.

[6] E. Cambouropoulos, M. Crochemore, C. Iliopoulos, L. Mouchard, and Y. J. Pinzon. Algorithms for computing approximate repetitions in musical sequences. In *Proc. AWOCA'99*, pages 129–144, 1999.

[7] E. Cambouropoulos, M. Crochemore, C. S. Iliopoulos, L. Mouchard, and Y. J. Pinzon. Algorithms for computing approximate repetitions in musical sequences. *J. of Computational Mathematics*, 79(11):1135–1148, 2002.

[8] P. Clifford, R. Clifford, and C. Iliopuolos. Faster algorithms for (δ, γ)-matching and related problems. In *Proc. CPM'05*, LNCS v. 3537, pages 68–78, 2005.

[9] R. Cole and R. Hariharan. Verifying candidate matches in sparse and wildcard matching. In *Proc. STOC'02*, pages 592–601, 2002.

[10] R. Cole, C. Iliopoulos, T. Lecroq, W. Plandowski, and W. Rytter. On special families of morpishms related to δ-matching and don't care symbols. *Information Processing Letters*, 85(5):227–233, 2003.

[11] T. Crawford, C. Iliopoulos, and R. Raman. String matching techniques for musical similarity and melodic recognition. *Computing in Musicology*, 11:73–100, 1998.

[12] M. Crochemore, C. Iliopoulos, T. Lecroq, Y. J. Pinzon, W. Plandowski, and W. Rytter. Occurence and substring heuristics for δ-matching. *Fundamenta Informaticae*, 55:1–15, 2003.

[13] M. Crochemore, C. Iliopoulos, G. Navarro, Y. Pinzon, and A. Salinger. Bit-parallel (δ, γ)-matching suffix automata. *J. of Discrete Algorithms*, 3(2–4):198–214, 2005.

[14] M. Crochemore, C. Iliopoulos, Y. Pinzon, and J. Reid. A fast and practical bit-vector algorithm for the longest common subsequence problem. *Information Processing Letters*, 80(6):279–285, 2001.

[15] M. Crochemore and W. Rytter. *Text algorithms*. Oxford University Press, 1994.

[16] S. Deorowicz. Speeding up transposition invariant string matching. Technical report, Institute of Computer Science, Silesian University of Technology, Poland, 2005. `http://www-zo.iinf.polsl.gliwice.pl/~sdeor/pub/deo05babs.htm`.

[17] K. Fredriksson and G. Navarro. Average-optimal single and multiple approximate string matching. *ACM J. of Experimental Algorithmics*, 9(1.4), 2004.

[18] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[19] H. Hyyrö, Y. Pinzon, and A. Shinohara. New bit-parallel algorithm for approximate string matching under indel distance. In *Proc. WEA'05*, LNCS v. 3503, pages 380–390, 2005.

[20] K. Lemström, G. Navarro, and Y. Pinzon. Practical algorithms for transposition-invariant string-matching. *J. of Discrete Algorithms*, 3(2–4):267–292, 2005.

[21] K. Lemström and E. Ukkonen. Including interval encoding into edit distance based music comparison and retrieval. In *Proc. AISB'00*, pages 53–60, 2000.

[22] V. Mäkinen. Sub-quadratic algorithm for weighted k-mismatches problem. Technical Report C-2004-1, Dept. of Computer Science, Univ. of Helsinki, 2004. `http://www.cs.helsinki.fi/u/vmakinen/papers/weightedkmm.ps.gz`.

[23] V. Mäkinen, G. Navarro, and E. Ukkonen. Transposition invariant string matching. *J. of Algorithms*, 2004. To appear. Conference version in *Proc. STACS'03*, LNCS 2607, pages 191–202.

[24] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

[25] G. Navarro and K. Fredriksson. Average complexity of exact and approximate multiple string matching. *Theoretical Computer Science*, 321(2–3):283–290, 2004.

[26] G. Navarro, Sz. Grabowski, V. Mäkinen, and S. Deorowicz. Improved time and space complexities for transposition invariant string matching. Technical Report TR/DCC-2005-4, Dept. of Computer Science, Univ. of Chile, 2005. `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/mnloglogs.ps.gz`.

[27] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings*. Cambridge University Press, 2002.

[28] P. Roland and J. Ganascia. Musical pattern extraction and similarity assessment. In E. Miranda, editor, *Readings in Music and Artificial Intelligence*, pages 115–144. Harwood Academic Publishers, 2000.

[29] A. C. Yao. The complexity of pattern matching for a random string. *SIAM J. of Computing*, 8(3):368–387, 1979.

Approximation Algorithm for the Cyclic Swap Problem

Yoan José Pinzón Ardila[1], Costas S. Iliopoulos[1], Gad M. Landau[2], and Manal Mohamed[1]

[1] King's College London, Dept. of Computer Science, London WC2R 2LS, UK
e-mail: Yoan.Pinzon@kcl.ac.uk, e-mail: ⟨csi,manal⟩@dcs.kcl.ac.uk

[2] Dept. of Computer Science, Haifa University, Haifa 31905, Israel
e-mail: landau@cs.haifa.ac.il

Abstract. Given two n-bit (cyclic) binary strings, A and B, represented on a circle (necklace instances). Let each sequence have the same number k of 1's. We are interested in computing the cyclic swap distance between A and B, *i.e.*, the minimum number of swaps needed to convert A into B, minimized over all rotations of B. We show that this distance may be approximated in $O(n + k^2)$ time.

1 Introduction

Cyclic string comparison is important for different domains where linear strings represent cyclic sequences, for example, in computational biology the genetic material is sequenced from circular DNA or RNA molecules. Bacterial, chloroplasts and mitochondrial genomes are in majority circular [2]. Small circular DNA molecules that have the ability to replicate on their own, are extensively used in biotechnology [2]. All such cyclic molecules are represented as linear strings by choosing an arbitrary starting point. It follows that the comparison of two such sequences needs to consider all possible cyclic shifts of one of the sequences. DNA, as well as RNA, are oriented molecules, therefore, in some cases, *e.g.*, for Expressed Sequence Tags, sequences must be compared in each orientation [1]. Other domains are pattern representation and recognition [6]. There, polygonal shapes are encoded into linear strings by choosing arbitrarily a start position on the contour. Determining if two shapes are similar requires to compare one string with all cyclic shifts of the other.

Another domain in which cyclic strings arise is computational music analysis. Mathematics and music theory have a long history of collaboration dating back to at least Pythagoras [11]. More recently the emphasis has been mainly on analysing string pattern matching problems that arise in music theory [7, 8, 9, 10]. A fundamental problem in music theory is to measure the similarity between rhythms, with many applications such as copyright infringement resolution and music information retrieval.

Six examples of 4/4 time clave and bell timelines are given in Fig. 1. The left-hand side shows the rhythms with standard Western music notation using the smallest convenient notes and rests. The right-hand side shows a popular way of representing

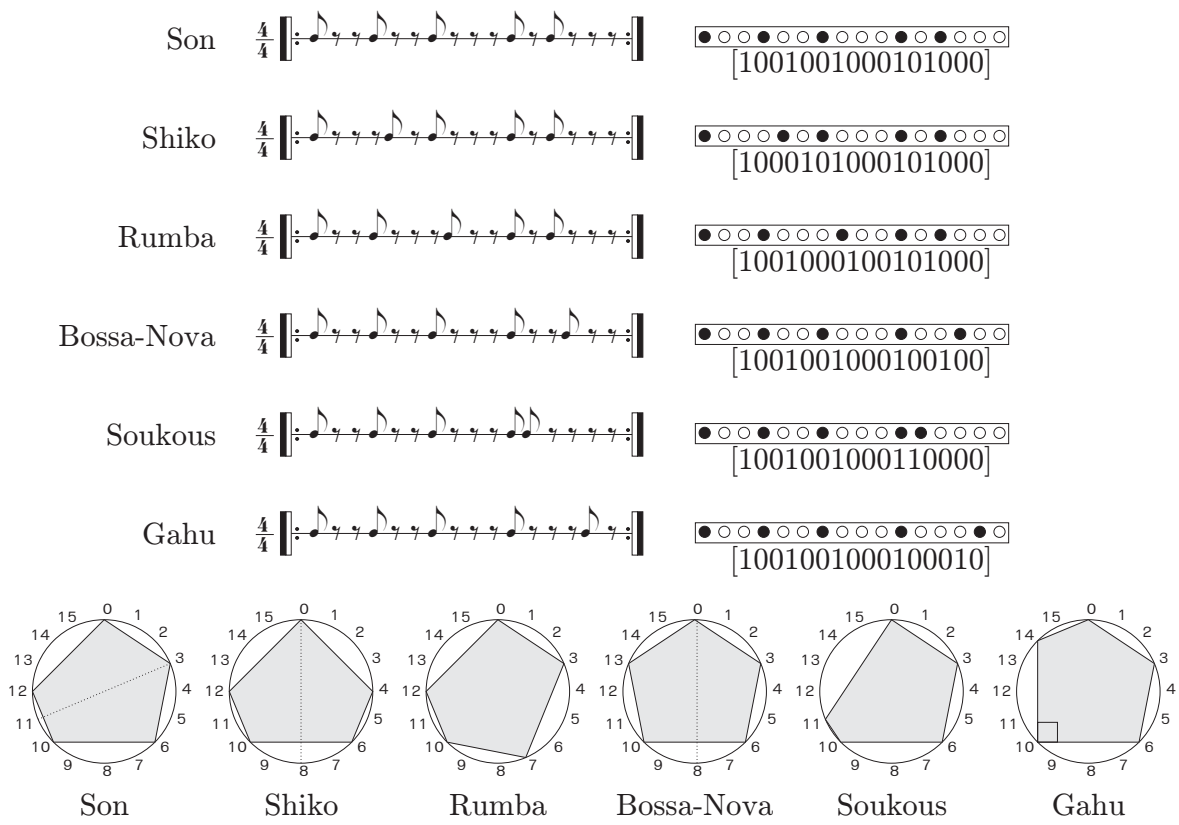


Figure 1: Six fundamental 4/4 time *clave* rhythms. The left-hand side uses the Western music notation and the right-hand size the box notation and binary representation. The bottom line shows a common geometric representation using convex polygons. The dashed lines indicate an axis of mirror symmetry (*e.g.* for the *son clave*, if the rhythm is started at location 3 then it sounds the same whether it is played forward or backwards).

rhythms for percussionists that do not read music. It is called the Box Notation Method developed by Philip Harland at the University of California in Los Angeles in 1962 and is also known as TUBS (Time Unit Box System). The box notation method is convenient for simple-to-notate rhythms like bell and clave patterns as well as for experiments in the psychology of rhythm perception, where a common variant of this method is simply to use one symbol for the note and another for the rest. The *Clave Son* is the most popular among these rhythms and can be heard a lot in Son and Salsa music as well as much other music around the world. For our purpose, A rhythm is represented as a cyclic binary sequence where a zero denotes a rest (silence) and a one represents a beat or note onset, for example, the clave Son would be written as the 16-bit binary sequence: [1001001000101000]. This rhythm can also be thought as a point in a 16-dimensional space (the hypercube). A natural measure of the difference between two rhythms represented as binary sequences is the well known Hamming distance, which counts the number of positions in which the two rhythms disagree. Although the Hamming distance measures the existence of a mismatch, it does not measure how far the mismatch occurs, that is why, Toussaint [15] proposed a distance measure termed the *swap distance*. A swap is an interchange

of a one and a zero (note duration and rest interval) that are adjacent in the sequence. The swap distance between two rhythms is the minimum number of swaps required to convert one rhythm to the other. The swap distance measure of dissimilarity was shown in [14] to be more appropriate than several other measures of rhythm similarity including the Hamming distance, the Euclidean interval-vector distance, the interval-difference distance measure of Coyle and Shmulevich, and the chronotonic distance measures of Gustafson and Hofmann-Engl.

More formally, given two n-bit (cyclic) binary strings, A and B, represented on a circle (necklace instances). Let each sequence have the same number k of 1's. We are interested in computing the cyclic swap distance between A and B, *i.e.*, the minimum number of swaps needed to convert A into B, minimized over all rotations of B. We show that this distance may be computed in $O(n + k^2)$.

The outline of the paper is as follows: Some preliminaries are described in Section 2. A Naive Solution is presented in Section 3 and in Section 4 we present a better-than-naive solution. Conclusions are drawn in Section 5.

2 Preliminaries

Let $X[0..n-1]$ be a n-bit *cyclic string* over $\Sigma = \{0, 1\}$, with $n \geq 0$. By $X[i]$ we denote the $(i+1)$-st bit in X, $0 \leq i <$length(X). Let ℓ=ones(X) be the number of 1's in X. Let X^r be the r-rotation of X such that $X^r[i] = X[i \circledplus r]$ for $i \in [0..n-1]$, and any integer r ($i \circledplus r = \mathrm{mod}(i + r, n)$[1]).

Let x be the increasing sequence of 1's indices $(x_0, x_1, \ldots, x_{\ell-1})$ such that $X[x_i] = 1$ for $i \in [0..\ell - 1]$. For $u = (u_0, u_1, \ldots, u_n)$, $v = (v_0, v_1, \ldots, v_n)$ and some integer e, we denote by $u \cdot v = (u_0, u_1, \ldots, u_n, v_0, v_1, \ldots, v_n)$ the *sequence concatenation operation* and by $u + e = (u_0 + e, u_1 + e, \ldots, u_n + e)$ the *sequence transposition operation.*

Given X and Y, two n-bit (cyclic) binary strings with the same number of 1's, the *cyclic swap problem* is to find the cyclic swap distance between X and Y, *i.e.*, the minimum number of swaps needed to convert X into Y, minimized over all rotations of Y. A swap is an interchange of a one and a zero that are adjacent in the binary string.

2.1 Mappings and Rotations

A mapping is a bijection function $\mathcal{M} : x \to y$. Since no two adjacent 1's can be swaped, no two mappings should cross. Hence, there are ℓ possible mappings. For instance, if $\ell = 3$, we could have the following three mappings $\{(x_0 \to y_0), (x_1 \to y_1), (x_2 \to y_2)\}$, $\{(x_0 \to y_1), (x_1 \to y_2), (x_2 \to y_0)\}$ and $\{(x_0 \to y_2), (x_1 \to y_0), (x_2 \to y_1)\}$. So we define

$$\mathcal{M}^k = \{\ (x_i \to y_{i \circledplus k})\ \mid\ i, k \in [0..\ell - 1]\ \} \tag{1}$$

to be the *k-mapping* of the 1's in X with those 1's in Y.

We also redefine y as $y \cdot (y + n)$ and denote

[1] We use operator $\circledplus$ to indicate that all indices are viewed modulo n.

$$\mathcal{D}^{(k,r)} = \sum_{i=0}^{\ell-1} |y_{i+k} - x_i + r|, \text{ for } k \in [0..\ell-1] \tag{2}$$

to be the sum of the number of swap between the pairs in $\mathcal{M}^k$ and rotation r of Y. (Here $|x|$ designates the absolute value of x.) The reason for using $y \cdot (y+n)$ instead of y is because M^k will always have k mappings that are circular, for example, for $\mathcal{M}^1$ we map x_2 with y_0 but the number of swap needed to map x_2 with y_0 is $(y_0+n) - x_2$ instead of $y_0 - x_2$. Fig. 2 shows rotations $-7, -6, \ldots, 6, 7$ for mapping $\mathcal{M}^0$ of $X = [10010001]$ and $Y = [01010010]$. Note that $\mathcal{D}^{(k,n-r)} + \mathcal{D}^{(k,-r)} = \ell n$ for $0 < r < n$.

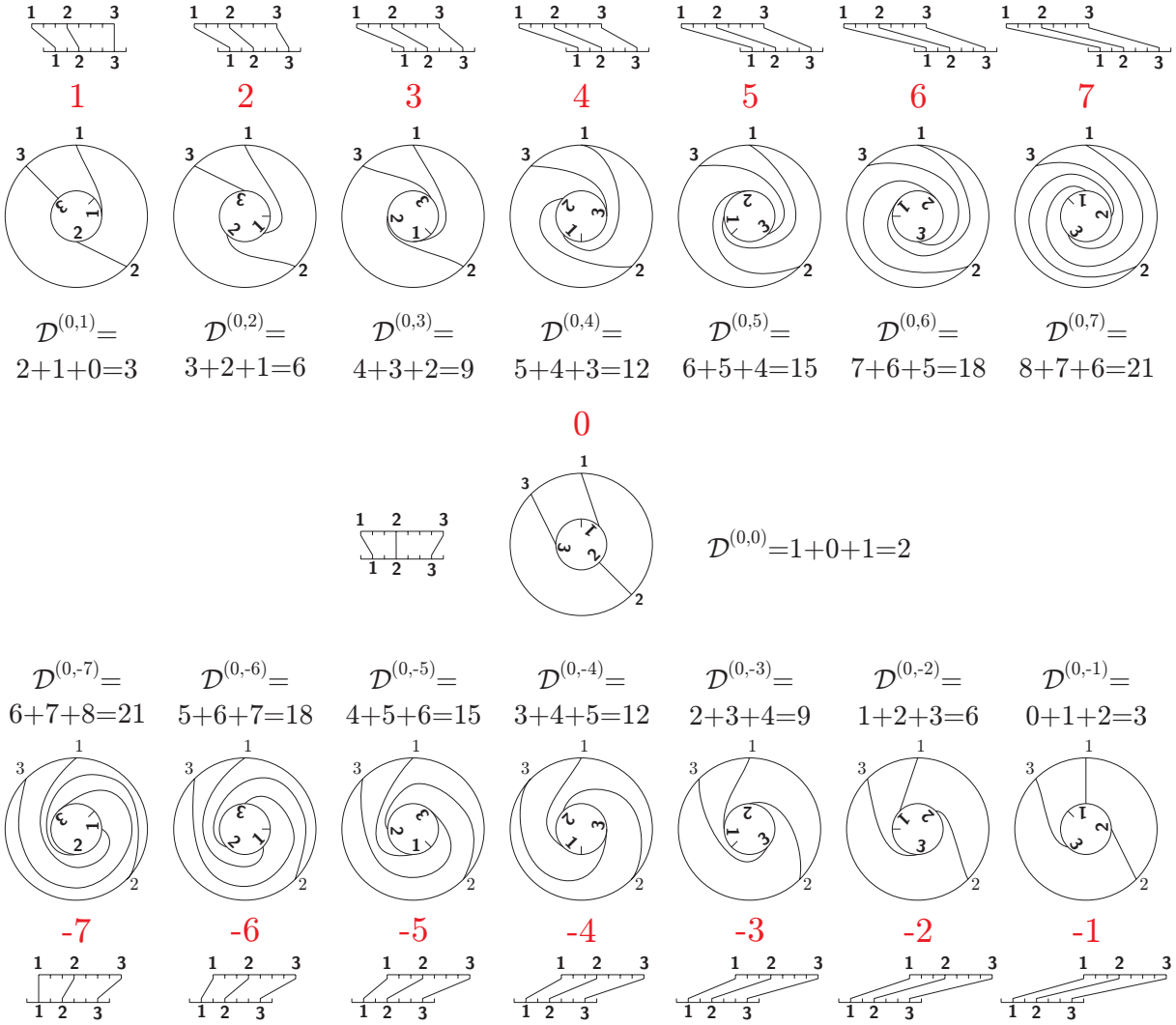


Figure 2: Rotations $-7, -6, \ldots, 6, 7$ for mapping $\mathcal{M}^0$ of $X = [10010001]$ and $Y = [01010010]$ or $x = (0, 3, 7)$ and $y = (1, 3, 6)$. X/Y corresponds to the outer/inner cyclic string.

3 Naive Solution

The naive approach is to examine each mapping and calculate for each possible rotation the sum of number of swap operations between each pair of mapped 1's. This approach costs $O(n\ell^2)$ time. This is because there are ℓ possible mappings and n possible rotations per each mapping. Fig. 3 shows the main steps of the algorithm.

Algorithm 1 Naive Solution

Input: x, y, n, ℓ
Output: $d_{min}, r_{min}, k_{min}$
1. $y = y \cdot (y + n)$; $d_{min} = r_{min} = k_{min} = \infty$
2. **for** $k = 0$ **to** $\ell - 1$ **do**
3. **for** $r = 0$ **to** $n - 1$ **do**
4. $d = 0$
5. **for** $i = 0$ **to** $\ell - 1$ **do**
6. $d = d + |y_{i+k} - x_i + r|$
7. **if** $d < d_{min}$ **then** $d_{min} = d, r_{min} = r, k_{min} = k$
8. **return** $(d_{min}, r_{min}, k_{min})$

Figure 3: Naive Algorithm.

4 Better-than-Naive Solution

The problem is reduced to find, for each mapping $\mathcal{M}^k$, the rotation r that minimizes Equation 2. This can be approximated by replacing the absolute values in (2) by squares so we get

$$\mathcal{D}'^{(k,r)} = \sum_{i=0}^{\ell-1} (y_{i+k} - x_i + r)^2, \text{ for } k \in [0..\ell - 1].$$

Lets say that $s_i^k = y_{i+k} - x_i$ so we get

$$\mathcal{D}'^{(k,r)} = \sum_{i=0}^{\ell-1} (s_i^k + r)^2,$$

which we can be rewritten as

$$\mathcal{D}'^{(k,r)} = \sum_{i=0}^{\ell-1} (s_i^k)^2 + 2r \sum_{i=0}^{\ell-1} s_i^k + \ell r^2. \tag{3}$$

Differentiating (3) and setting the result equal to zero, we obtain

$$\frac{\partial \mathcal{D}'^{(k,r)}}{\partial r} = 2 \sum_{i=0}^{\ell-1} s_i^k + 2\ell r = 0. \tag{4}$$

If we say that $\mathcal{S}^k = \sum_{i=0}^{\ell-1} s_i^k$ and solving r from (4) it follows $r = -\mathcal{S}^k/\ell$. The corollary good news is that

$$\mathcal{S}^k = \mathcal{S}^{k-1} + n, \text{ for } k \in [2..\ell-1]. \tag{5}$$

So, if we let

$$f = \mathcal{S}^0 = \sum_{i=0}^{\ell-1} s_i^0 = s_0^0 + s_1^0 + \cdots + s_{\ell-1}^0 = y_0 + y_1 + \cdots + y_{\ell-1} - x_0 - x_1 - \cdots - x_{\ell-1},$$

then

$$r'_k = -\frac{f + kn}{\ell}, \text{ for } k \in [0..\ell-1], \tag{6}$$

will correspond to the rotation that will minimize the square of the swap distance for mapping $\mathcal{M}^k$. Thus, $\mathcal{D}'^{(k,r'_k)}$ is optimal for $\mathcal{M}^k$.

Now that we know how to find the best rotation r'_k for each mapping, it is easy to compute the minimum swap distance for each such rotation and the best overall rotation will be given by

$$d^* = \min_{i=0}^{\ell-1}(\mathcal{D}'^{(i,r'_i)}).$$

The time complexity to compute f is $\Theta(\ell)$ and each r'_i for $i \in [0..\ell-1]$ can be compute in $\Theta(1)$ time using Equation 6. So the time required to compute all rotations $r'_0, r'_1, \ldots, r'_{\ell-1}$ is $\Theta(\ell)$. Once we have the rotations, the time to compute the swap distance for each mapping and each rotation $\mathcal{D}^{(k,r'_k)}$, for $k \in [0..\ell-1]$, is $\Theta(\ell)$ if done naively. Hence, the total time to find d^* is $\Theta(\ell^2)$ or $\Theta(n + \ell^2)$ if we assume that sequences x and y are to be computed. Fig. 4 shows these ideas algorithmically.

Algorithm 2 Better-than-Naive Solution

Input: x, y, n, ℓ
Output: $d_{min}, r_{min}, k_{min}$
1. $y = y \cdot (y + n)$; $f = 0$; $d_{min} = r_{min} = k_{min} = \infty$
2. **for** $i = 0$ **to** $\ell - 1$ **do**
3. $\quad f = f + y_i - x_i$
4. **for** $k = 0$ **to** $\ell - 1$ **do**
5. $\quad r = -(f + nk)/\ell$; $d = 0$
6. $\quad$ **for** $i = 0$ **to** $\ell - 1$ **do**
7. $\quad\quad d = d + |y_{i+k} - x_i + r|$
8. $\quad$ **if** $d < d_{min}$ **then** $d_{min} = d, r_{min} = r, k_{min} = k$
9. **return** $(d_{min}, r_{min}, k_{min})$

Figure 4: Better-than-Naive Algorithm.

Note that the value d^* is an approximated cyclic swap distance. Our experiments show that in very few cases d^* was not the optimal cyclic swap distance. However, it

indicates correctly the optimal mapping $\mathcal{M}^*$. Additionally, we were also able to prove [4] that if r_k was the optimal exact rotation for mapping $\mathcal{M}^k$, then at least one of the values $|y_{i+k} - x_i + r_k|$ is equal to 0, for $0 \leq i \leq \ell - 1$. Thus, the exact cyclic swap distance may be calculated using first Algorithm 4 to calculate the optimal mapping. Then, if none of the values $|y_{i+k} - x_i + r|$ (Line 8) is equal to 0, then additional $O(k^2)$ is needed to calculate the optimal cyclic swap distance. Clearly, this is not going to effect the overall running time.

Before continuing our discussion, we show why Equation 5 is correct. First, we illustrate the formula using the example in Fig. 5. Here, $f = \mathcal{S}^0 = s_0^0 + s_1^0 + s_2^0 = 1-2-3 = -4$, $\mathcal{S}^1 = s_0^1 + s_1^1 + s_2^1 = 3+0+3 = 6$ and $\mathcal{S}^2 = s_0^2 + s_1^2 + s_2^2 = 5+6+5 = 16$. However, we do not need to compute $s_0^1, s_1^1, s_2^1, s_0^2, s_1^2$ and s_2^2 in order to compute $\mathcal{S}^1$ and $\mathcal{S}^2$, instead, we apply Equation 5 and find that $\mathcal{S}^1 = \mathcal{S}^0 + n = -4 + 10 = 6$ and $\mathcal{S}^2 = \mathcal{S}^0 + 2n = -4 + 20 = 16$.

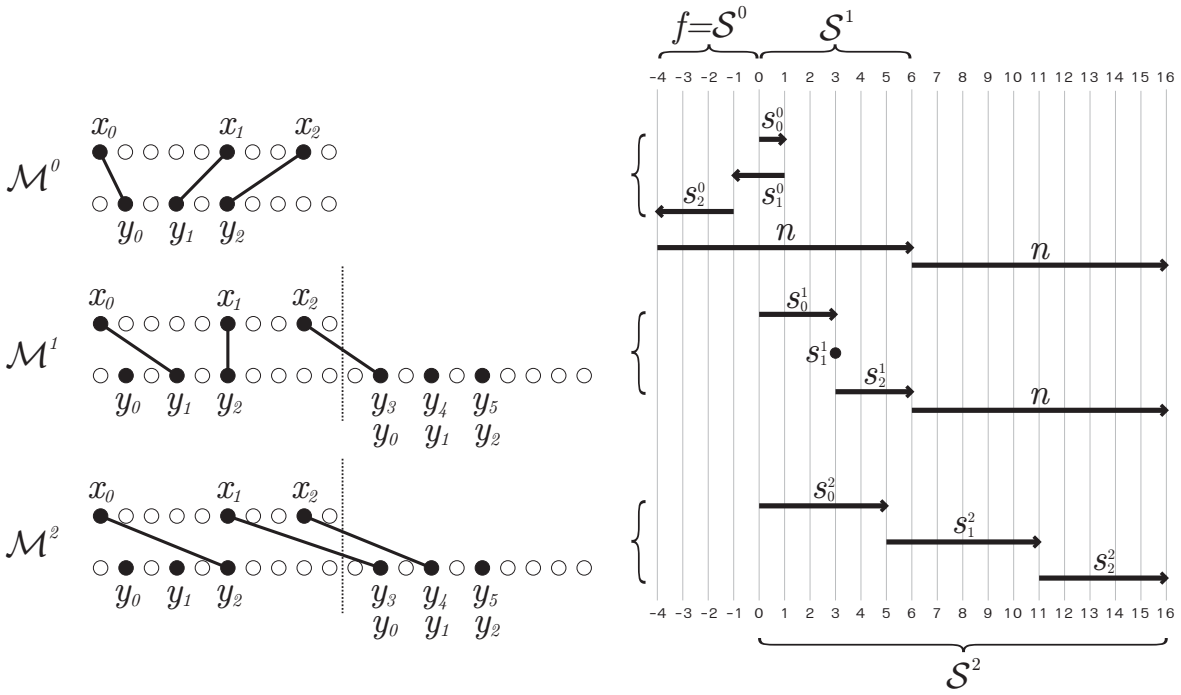


Figure 5: Illustration to demonstrate the correctness of Equation 5 for X = [1000000100000101] and Y = [0001000000101010] or $x = (0, 5, 8)$ and $y = (1, 3, 5)$. Note that $\mathcal{S}^1 = \mathcal{S}^0 + n$ and $\mathcal{S}^2 = \mathcal{S}^1 + n = \mathcal{S}^0 + 2n$.

More formally we know that for $\mathcal{M}^k$

$$\mathcal{S}^k = \sum_{i=0}^{\ell-1} s_i^k = \sum_{i=0}^{\ell-1} (y_{i+k} - x_i) = \sum_{i=0}^{\ell-1} y_{i+k} - \sum_{i=0}^{\ell-1} x_i.$$

Note that

$$\begin{aligned}\sum_{i=0}^{\ell-1} y_{i+k} &= y_{0+k} + y_{1+k} + ... + y_{\ell-2+k} + y_{\ell-1+k} \\ &= y_{1+(k-1)} + y_{2+(k-1)} + ... + y_{\ell-1+(k-1)} + y_{k-1} + n \\ &= y_{0+(k-1)} + y_{1+(k-1)} + y_{2+(k-1)} + ... + y_{\ell-1+(k-1)} + n \\ &= \sum_{i=0}^{\ell-1} y_{i+(k-1)} + n.\end{aligned}$$

Therefore, Equation 5 is correct.

Example. Lets assume we want to solve the cyclic swap problem for the cyclic binary strings $X = [1000000100000101]$ and $Y = [0001000000101010]$. Then $n = 16$, $\ell = 4$, $x = (0, 7, 13, 15)$ and $y = (3, 10, 12, 14)$. Recall we double y by adding $(y + n)$ to be able to compute the initial mappings, so y will be $(3, 10, 12, 14, 19, 26, 28, 30)$. (See Fig. 6 for a visualization of the input data.)

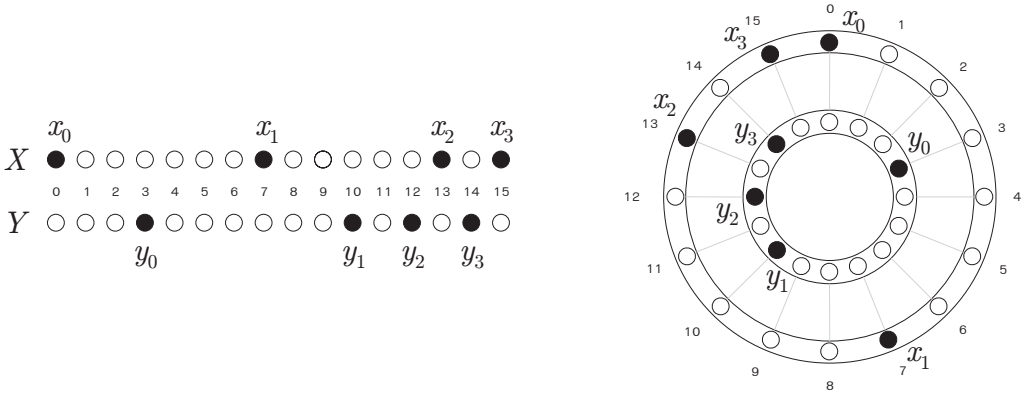


Figure 6: Example for $x = (0, 7, 13, 15)$ and $y = (3, 10, 12, 14)$. The right-hand side depicts the strings circularly.

The first step is to compute f as follows

$$\begin{aligned}f &= \sum_{i=0}^{3} s_i^0 = s_0^0 + s_1^0 + s_2^0 + s_3^0 \\ &= y_0 + y_1 + y_2 + y_3 - x_0 - x_1 - x_2 - x_3 \\ &= 3 + 10 + 12 + 14 - 0 - 7 - 13 - 15 \\ &= 4.\end{aligned}$$

Now we compute the best rotation for each mapping applying Equation 6 and get

$$r'_0 = -\frac{4}{4} = -1, \quad r'_1 = -\frac{4+16}{4} = -5, \quad r'_2 = -\frac{4+32}{4} = -9, \quad r'_3 = -\frac{4+48}{4} = -13.$$

The next step is to find the swap distance for each mapping using Equation 2 and the r's we found in the previous step

$$\begin{aligned}
\mathcal{D}^{(0,r'_0)} &= |y_0 - x_0 + r_0| + |y_1 - x_1 + r_0| + |y_2 - x_2 + r_0| + |y_3 - x_3 + r_0| \\
&= |3 - 0 - 1| + |10 - 7 - 1| + |12 - 13 - 1| + |14 - 15 - 1| \\
&= 2 + 2 + 2 + 2 = 8.
\end{aligned}$$

$$\begin{aligned}
\mathcal{D}^{(1,r'_1)} &= |y_1 - x_0 + r_1| + |y_2 - x_1 + r_1| + |y_3 - x_2 + r_1| + |y_4 - x_3 + r_1| \\
&= |10 - 0 - 5| + |12 - 7 - 5| + |14 - 13 - 5| + |19 - 15 - 5| \\
&= 5 + 0 + 4 + 1 = 10.
\end{aligned}$$

$$\begin{aligned}
\mathcal{D}^{(2,r'_2)} &= |y_2 - x_0 + r_2| + |y_3 - x_1 + r_2| + |y_4 - x_2 + r_2| + |y_5 - x_3 + r_2| \\
&= |12 - 0 - 9| + |14 - 7 - 9| + |19 - 13 - 9| + |26 - 15 - 9| \\
&= 3 + 2 + 3 + 2 = 10.
\end{aligned}$$

$$\begin{aligned}
\mathcal{D}^{(3,r'_3)} &= |y_3 - x_0 + r_3| + |y_4 - x_1 + r_3| + |y_5 - x_2 + r_3| + |y_6 - x_3 + r_3| \\
&= |14 - 0 - 13| + |19 - 7 - 13| + |26 - 13 - 13| + |28 - 15 - 13| \\
&= 1 + 1 + 0 + 0 = 2.
\end{aligned}$$

Fig. 7 shows the best rotations for each mapping. We conclude that mapping $\mathcal{M}^3$ with Y rotated by -13 (+3) gives the best swap distance $\mathcal{D}^{(3,-13)} = \mathcal{D}^{(3,+3)} = 2$.

5 Conclusions

We have presented a new algorithm that solve the problem of cyclic swap distance between two n-bit (cyclic) binary strings in $O(n + \ell^2)$ where ℓ is the number of 1's (same) in both strings.

The fact that a n-bit binary string could be thought as a point in a n-dimensional space (the hypercube) suggests the strong links between the cyclic swap problem and the problem of calculating the closest pairs problem in high dimension. The later problem is a fundamental and well-studied problem in computational geometry. For dimension $d = n$ (which is the case here), Shamos and Bently [5] conjectured that the problem can be solved in $O(n^2 \log n)$ time, where n is the number of points all in $\mathcal{R}^d$. Recently, a better-than-naive-solution has been presented with running time of $O(n^{(\omega+3)/2})$, where $O(n^\omega)$ is the running time of matrix multiplication [13]. Several approximate algorithms were designed for the high-dimensional closest pair problem; see [12] for a survey of such algorithms. We plan to use some of these ideas, plus the fact that the swap distance problem is equivalent to calculate the L_1 distance [3], to further improve the time complexity of the presented algorithm, where for two strings X, Y, $L_1(X, Y)$ is defined as follows:

$$L_1(X, Y) = \sum_{i=0}^{n-1} |Y[i] - X[i]|.$$

The question of whether the swap distance can be calculated in more efficient time is left as an open problem.

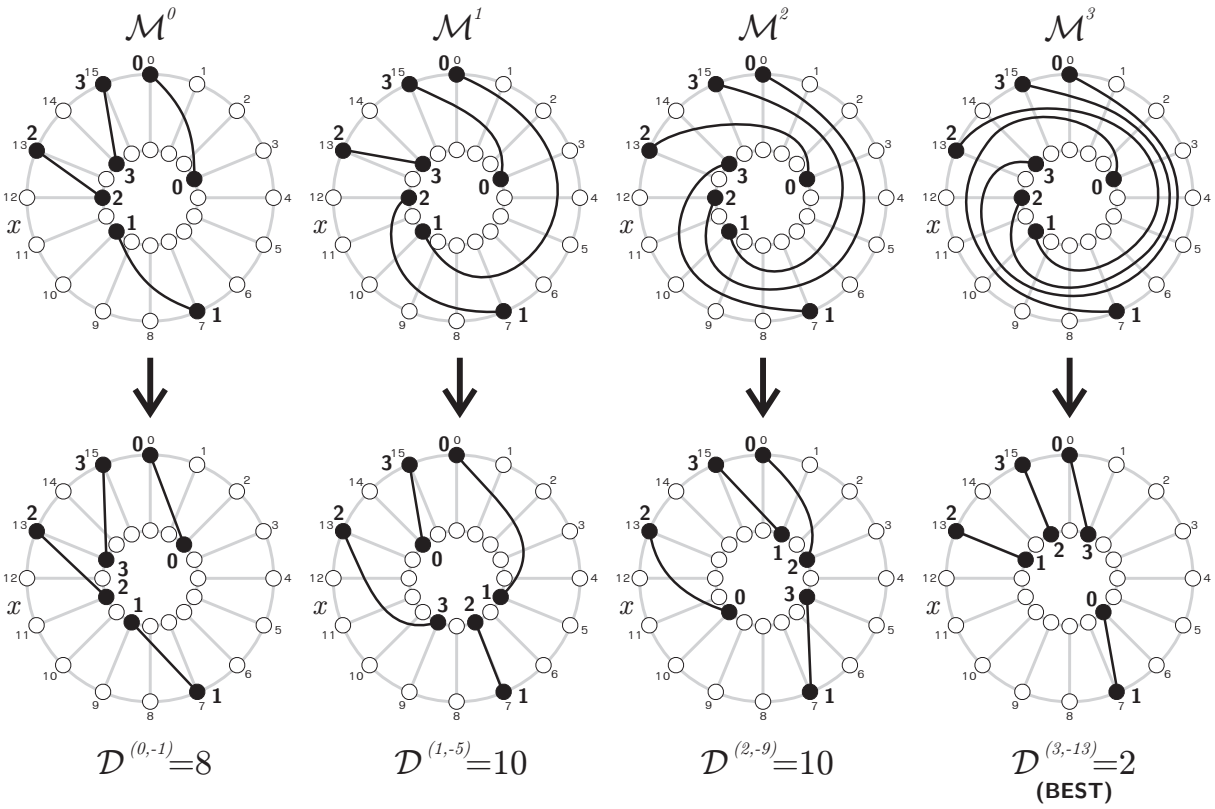


Figure 7: Computation of the minimum cyclic swap distance for $X = [1000000100000101]$ and $Y = [0001000000101010]$ or $x = (0, 7, 13, 15)$ and $y = (3, 10, 12, 14)$. The inner circle represents string X while the outer circle represents string Y. The first row, shows the initial mappings corresponding to Equation 1. The second row shows the result after shifting the original mappings by the rotations computed using Equation 6, thus mapping $\mathcal{M}^0$ is shifted by $r'_0 = -1$ (a negative value means that the shift is performed anti-clockwise), $\mathcal{M}^1$ by -5, $\mathcal{M}^2$ by -9, and $\mathcal{M}^3$ by -13. For each new rotation the corresponding swap distance was calculated using Equation 2 and it can be seen that the best result is given by $\mathcal{M}^3$ with an anti-clockwise rotation of 13.

References

[1] M. D. Adams, J. M. Kelley, J. D. Gocayne, M. Dubnick, M. H. Polymeropoulos, H. Xiao, C. R. Merril, A. Wu, B. Olde, R. F. Moreno, et al. Complementary DNA sequencing: expressed sequence tags and human genome project. *Science*, 252(5013):1651–1656, 1991.

[2] B. Alberts, D. Bray, J. Lewis, M. Raff, K. Roberts, and J. D. Watson. *Molecular Biology of the Cell*. Garland, New York, 1983.

[3] A. Amir, O. Lipsky, E. Porat and J. Umanski. Approximate Matching in the L_1 Metric. *In Proceedings of the 16th Annual Symposium on Combinatorial Pattern Matching (CPM'05)*, pp. 91–103, 2005.

[4] Y. J. Pinzón Ardila, R. Clifford and M. Mohamed. Necklace Swap Problem for Rhythmic Similarity Measures. Submitted for publication.

[5] J. L. Bentley and M. I. Shamos. Divide-and-Conquer in Multidimensional Space. *STOC*, pp. 220–230, 1976.

[6] H. Bunke and U. Buehler. Applications of approximate string matching to 2D shape recognition. *Pattern Recognition*, 26(12):1797—1812, 1993.

[7] E. Cambouropoulos, M. Crochemore, C. S. Iliopoulos, L. Mouchard, and Y. J. Pinzon. Computing approximate repetitions in musical sequences. *International Journal of Computer Mathematics*, 79(11):1135–1148, 2002.

[8] R. Clifford, T. Crawford, C. Iliopoulos, and D. Meredith. Problems in computational musicology. In C. S. Iliopoulos and Thierry Lecroq, editors, *String Algorithmics*, NATO Book series, King's College Publications, 2004.

[9] T. Crawford, C. S. Iliopoulos, R. Raman, String Matching Techniques for Musical Similarity and Melodic Recognition. *Computing in Musicology*, 11:71–100, 1998.

[10] M. Crochemore, C. S. Iliopoulos, G. Navarro, and Y. Pinzon. *A bit-parallel suffix automaton approach for (δ,γ)-matching in music retrieval.* In M. A. Nascimento, Edleno S. de Moura, and A. L. Oliveira, editors, 10th International Symposium on String Processing and Information Retrieval, SPIRE 2003, Springer-Verlag, pp. 211–223, 2003.

[11] J. Godwin, *The Harmony of the Spheres: A Sourcebook of the Pythagorean Tradition in Music*, Inner Traditions Intl. Ltd, 1993.

[12] P. Indyk. Nearest neighbors in high-dimensional spaces. In Jacob E. Goodman and Joseph O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 39. CRC Press, 2rd edition, 2004.

[13] P. Indyk, M. Lewenstein, O. Lipsky, E. Porat. Closest Pair Problems in Very High Dimensions. *ICALP* pp. 782–792, 2004.

[14] G. T. Toussaint, *Computational geometric aspects of musical rhythm*, Abstracts of the 14th Annual Fall Workshop on Computational Geometry, Massachussetts Institute of Technology, November 19-20, pp. 47–48, 2004.

[15] G. T. Toussaint, *A comparison of rhythmic similarity measures*, Proceedings of ISMIR 2004: 5th International Conference on Music Information Retrieval, Universitat Pompeu Fabra, Barcelona, Spain, October 10-14, 2004, pp. 242–245. A longer version also appeared in: School of Computer Science, McGill University, Technical Report SOCS-TR-2004.6, 2004.

Incremental String Correction: Towards Correction of XML Documents

Ahmed Cheriat*, Agata Savary†, Béatrice Bouchou, and Mírian Halfeld Ferrari

Université François Rabelais de Tours - LI/Campus de Blois, France
3 place Jean Jaurès - 41000 Blois, France

ahmed.cheriat@etu.univ-tours.fr
{agata.savary, beatrice.bouchou, mirian}@univ-tours.fr

Abstract. We define a problem of an *incremental* string-to-string correction with respect to a regular grammar. A user is given a valid word which may be updated through one or more editing operations. If the resulting word is invalid we propose correction candidates that take not only the incorrect word but also the initial valid word into account. The method is based on the error distance matrix calculation as proposed by [9]. It has been developed in view of incremental XML document correction (as opposed to correction from scratch). Experimental results show a good performance of our algorithm despite its exponential theoretical complexity.

1 Introduction

We introduce an *incremental* string-to-string correction method with respect to a regular grammar. Given an initial correct (*valid*) word A (*i.e.* a word accepted by a regular grammar), a user can adapt this word to his needs by proposing one or more elementary operations (*updates*) on it under the condition that the resulting word B remains valid. If however B happens to be invalid (*e.g.* due to user's mistake when performing updates) the system should guess the user's intention and propose a set of plausible corrections. Thus, we are not willing to search for all nearest neighbors of B in the dictionary but only those that might result from A through a sequence of operations which are similar (but not identical) to the updates proposed by the user.

Our solution is to explore the finite-state automaton corresponding to the grammar in order to find valid words that are as close as possible to both A and B. Thus, we benefit from the achievements of the string-to-string correction domain ([11], [3]), as well as of their due to the finite-state representation of grammar or lexicon ([9]), while providing some new ideas focused on incrementality.

The motivation for the incremental string-to-string correction comes from the area of XML-document validation and correction. The validity of each node in such a document is described by one or more regular expressions. When a user wishes to

*Supported by Région Centre, France
†Partly supported by the IUT of Blois, France

modify a valid document but performs an invalid update on a node we may start with *locally* correcting this node's closest neighborhood using the incremental string-to-string approach. Thus, some good parts of the proposed correction tree may remain unchanged with respect to the initially valid XML-tree, which spares computation time and space.

As we place ourselves in a database context, updates are not treated one by one but grouped into sequences, or *transactions*. Thus, we are interested in the validity of the resulting word only at the end of each transaction. If the word turns out to be invalid we try to correct it with respect to the whole sequence of updates appearing in the transaction.

The paper is organized as follows. In Section 2, we resume some related work in the string-to-string correction domain. Then, in Section 3, we consider some particularities of our approach. Our incremental string correction method is described in Section 4. In Section 5 we discuss the complexity of our algorithm together with some experimental results. Finally, Section 6 concludes the paper, and gives some ideas of our future work.

2 String-to-String Correction with Respect to a Regular Grammar: State of the Art

The definition of the string-to-string correction problem aims at the formalization of the intuitive notion of similarity between two strings. As Wagner and Fischer ([11]) put it, the *edit distance* between two strings A and B is the minimum cost of all sequences of elementary edit operations (insertions, omissions and replacements) on letters which transform A into B. These operations may be written as rewriting rules of the form $a \to b$ where a and b are single letters and/or empty strings (ϵ) and $(a, b) \neq (\epsilon, \epsilon)$. Each edit operation $a \to b$ is assigned any non negative cost $\gamma(a \to b)$. We say that $a \to b$ takes A to B if $A = \sigma a \tau$ and $B = \sigma b \tau$.

An edit sequence S is a sequence $s_1, s_2, \ldots, s_m$, where s_i is an edit operation for each $0 \leq i \leq m$. Each edit operation s_{i+1} applies to the string resulting from the application of the preceding edit operation s_i. We say that S takes word A to word B if a sequence of strings $A_0, A_1, \ldots, A_m$ exists such that $A = A_0$, $B = A_m$ and s_i takes A_{i-1} to A_i for each $0 \leq i \leq m$. The cost $\gamma(S)$ of an edit sequence S is the sum of costs of all edit operations appearing in S.

Note that, with Wagner and Fischer, an edit sequence contains no reference to the word positions at which the edit operations operate. Due to this fact, the result of an edit sequence may be ambiguous. Moreover, a further edit operation may operate on a letter resulting from a former operation. For example, the application of the edit sequence $(a \to b, b \to c)$ to the word *abb* may result in any of the following words: *cbb*, *bcb*, *bbc*.

Furthermore, Wagner and Fischer ([11]) propose a useful model of a *trace* which is a visualization of a class of edit operation sequences as in the example on Figure 1. A line leading from position i of the source string A to position j of the target string B indicates that $A[i]$ should be replaced by $B[j]$ (if $A[i] \neq B[j]$) or that $A[i]$ should remain unchanged in B (if $A[i] = B[j]$). Characters of A untouched by any line are to be deleted and characters of B untouched by any line are to be inserted.

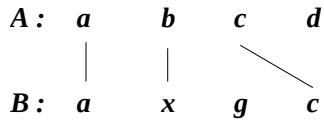


Figure 1: A trace between *abcd* and *axgc*

		ε	a	b	a	a	b	a
		0	1	2	3	4	5	6
ε	0	0	1	2	3	4	5	6
b	1	1	1	1	2	3	4	5
a	2	2	1	2	1	2	3	4
b	3	3	2	1	2	2	2	3

Figure 2: Edit distance matrix between *bab* and *abaaba*

Each trace T receives a non negative cost defined as follows:

$$\text{cost}(T) = \sum_{(i,j)\in T} (\gamma(A[i] \to B[j])) + \sum_{i\in I} (\gamma(A[i] \to \epsilon)) + \sum_{j\in J} (\gamma(\epsilon \to B[j]))$$

where I and J are the sets of positions in A and B, respectively, untouched by any line in T. For instance, if we assume that $\gamma(a \to b) = 1$ for $a \neq b$ then the trace on Figure 1 has cost 3.

It is further shown in [11] that a correspondence exists between edit sequences and traces:

- for every trace T from A to B, there is an edit sequence S taking A to B such that $\gamma(S) = \text{cost}(T)$
- for every edit sequence S taking A to B, there is a trace T from A to B such that $\text{cost}(T) \leq \gamma(S)$

Thus, looking for the minimum cost edit sequence taking A to B is equivalent to looking for the minimum cost trace from A to B. This minimum cost in both cases determines the edit distance between A and B. It can be obtained by a *dynamic programming* method which calculates an edit distance matrix H. For $0 \leq i \leq |A|$ and $0 \leq j \leq |B|$, element $H[i, j]$ contains the edit distance between the prefixes $A[1 : i]$ and $B[1 : j]$ of A and B (where $X[i : j]$ represents the subword $X_i, X_{i+1}, \ldots, X_{j-1}, X_j$). The matrix may be calculated column per column. Thus, each new element is deduced from its three top-left-hand neighbor elements which have been calculated previously. The bottom right-hand element of the whole matrix contains the edit distance between the two strings A and B. It is obtained with a time complexity of $O(|A| * |B|)$.

For example, the distance matrix on Figure 2 obtained by the above algorithm, with $\gamma(a \to b) = 1$ for $a \neq b$, indicates that the edit distance between strings *bab* and *abaaba* is 3.

Note that there may be more than one minimum cost trace (and thus more than one minimum cost edit sequence) between two words. In the above example, two such traces exist as shown on Figure 3.

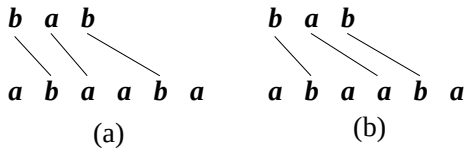


Figure 3: Two possible minimum cost traces between *bab* and *abaaba*

Lowrance and Wagner ([7]) extended the definition of the string-to-string correction problem to the case of four elementary editing operations on letters: the previous three operations were completed by a transposition of two adjacent letters. Thus, traces can contain crossing lines. However, the cost function was restricted to the case when all insertions, all deletions, all replacements, and all transpositions have the same costs W_I, W_D, W_C, W_S, respectively. An efficient solution ($O(|A| * |B|)$) for the edit distance calculation was proposed in case when $W_I + W_D \leq 2W_S$.

The addition of the transposition as the fourth elementary operation makes the mathematical model of the problem more complex. An elementary operation may still be represented as a rewriting rule of the type $\gamma(a \rightarrow b)$. However, the a and b symbols have now to be seen as sequences of letters rather than single letters. In the class of all possible sequences the choice of allowing only rules of type $xy \rightarrow yx$ seems very application-oriented. Note that with [7] the editing operations may still act on arbitrary positions in the source string, and in an arbitrary order (*e.g.* *ca* can be obtained from *abc* by two operations: deletion of b and transposition of a and c).

Du and Chang ([3]) modified this distance measure and renamed it to *error distance* by assigning cost 1 to each editing operation and by admitting that errors occur in linear order from left to right so that a later operation may not cancel the effect of an earlier operation. For example, two changes may not operate on the same word position while inversions occur only between letters that are adjacent in the original word and remain adjacent in the erroneous word (*e.g.* the error distance between *abc* and *ca* is 3). The linear order of editing operations in an edit sequence implies that each operation is assigned an integer corresponding to the current word position it operates on. For example, the edit sequence $(D(1), C(1, c), T(2))$ (*i.e.* deletion of letter at position 1, change of letter at position 1 to c, and transposition of letters at positions 2 and 3) applied to *abba* results in *cab*. Due to the equal cost of each editing operation, the error distance becomes a *metric*, *i.e.* a function satisfying four properties: non-negative values, reflexivity, symmetry, and triangular inequality.

The model simplification proposed by Du and Chang allows a substantial gain of efficiency to the algorithm of the error distance calculation. While in [7] the calculation of element $H[i, j]$ of the matrix needs, in the worst case, an access to each element of the previously calculated part of the matrix (to the left and above $H[i, j]$), with the linear error distance of Du and Chang this is no longer the case: $H[i, j]$ is calculated on the basis of its four neighbors only ($H[i-1, j]$, $H[i, j-1]$, $H[i-1, j-1]$, and $H[i-2, j-2]$). The matrix calculation has been further simplified due to some of its discovered properties.

Also in [3] the string-to-string correction is applied to the problem of finding, for a word, all its nearest neighbors in a dictionary. A distance threshold is one of the parameters of this problem. A nearest neighbor of X must stay within the error distance from X which is no bigger than the threshold. The dictionary is represented

in no particular form. A distance matrix has to be constructed from scratch for each new dictionary word with respect to the erroneous word. A cut-off criterion has been discovered which allows to stop the calculation of the matrix in its early stage as soon as it turns out that the error distance between two current strings exceeds the threshold. However, this calculation remains costly as it is roughly proportional to the number of words in the dictionary.

In the early applications of the approximate string matching ([4]), such as the automatized correction of computer programs, the vocabulary size was small (number of all key words and variable names in a program). Solutions as the one by [3] could then be applied with no problem of robustness.

As soon as the same string-to-string correction algorithms were to be used for spelling correction of natural language texts the vocabulary size often proved to be a bottleneck ([6]) which required additional dictionary reduction techniques. However, an extensive development of finite-state methods for natural language processing enabled a very time and space-efficient representation of large vocabularies. Furthermore, the dynamic programming method could be applied in the process of a finite-state dictionary access, thus providing a fast algorithm of searching for the nearest neighbors of a string in a dictionary. This technique was announced already by [10] for the 3-operation edit distance, but [9] was probably the first to extend it to the 4-operation error distance and test it extensively on large natural language vocabularies. In his algorithm, when a word is searched for in a finite-state lexicon, a part of the error matrix is calculated only once for all lexicon words that have the same common prefix. This optimization, in addition to the cut-off criterion of [3], provides an algorithm that rapidly finds, for a given word, all its t-distant neighbors in the dictionary.

More recent approaches to approximate string search in a finite-state dictionary, such as [8] which uses so-called Levenshtein automata and a "backward" dictionary, allow a further increase in speed of the string-to-string correction.

A string of symbols may be viewed as a trivial case of a tree whose depth is 1 and whose leaves are the elements of the string. Thus, the formalization of the string-to-string correction problem naturally inspired research on the tree-to-tree correction problem ([2]). Note that the diversity of the possible choices of elementary editing operations is even bigger in case of a tree as one can consider changes not only on the siblings' level but also on some ancestors' level. The most appropriate choice depends on the intuitive notion of tree proximity for the particular application. In our application, trees are XML documents which must be validated and corrected against their DTDs or XML schemata. However, compared to other tree correction approaches, our approach is to propose an *incremental* correction method as described in the following section.

3 Incremental String-to-String Correction with Respect to a Regular Grammar

The distance measure between two strings admitted in our approach is a simplified version of the edit distance by Wagner and Fischer ([11]) and of the error distance by Du and Chang ([3]). On the one hand, we allow only three elementary operations:

an insertion, a deletion, and a replacement of a single letter. On the other hand, we admit cost 1 for each of these operations.

The originality of our approach is due to three facts. Firstly, the definition of an edit operation (which we also call an *update*) and of an edit sequence (a *sequence of updates*) is particular. We attribute to each operation a word position it applies to as is the case with Du and Chang ([3]). However, all of these positions, numbered from 0 to the length of the word minus 1, concern the same initial word. For instance, the update sequence $(insert(a,0), replace(c,1), insert(d,3))$ takes the initial word abb to $aacbd$[1]. This definition of the word position is inspired by research on incremental XML validation by [1]. Note that this approach allows no later operation in a sequence to cancel the effect of an earlier operation, as is the case with [3].

Secondly, we place ourselves in a database context in which updates are not treated one by one but grouped into sequences, or *transactions*. That is because, given a sequence of n updates, a word may become incorrect after $i < n$ updates, but its validity may be re-established after all the n updates. For example, given the simple regular grammar $abcd + bced$, the initial valid word $abcd$, and the edit sequence $(delete(a,0), insert(e,3))$, the resulting word is valid ($bced$) and does not need any correction. If however we try to process the updates one by one we'll have to propose corrections for the intermediate invalid word bcd, which is useless for the user.

Thirdly, we wish to perform an *incremental* string-to-string correction in the context of a human-computer interaction. A user is given an initial correct word A (*i.e.* a word valid with respect to a regular grammar). He/she may adapt this word to his needs (or, in other words, construct a new word *incrementally*, or *evolutionarily*) by proposing one or more updates on this word under the condition that the resulting word B remains valid. If however B happens to be invalid (*e.g.* due to the user's ignorance with respect to the validity of words) the system should guess the user's intention and propose a set of plausible corrections. Thus, we are not willing to search for all nearest neighbors of B in the language described by the grammar but only those that might result from A through a sequence of operations which are similar (but not identical) to the updates proposed by the user. This approach, as opposed to a validation *from scratch* (where A is not taken into account), allows to possibly limit the computation time and space, as well as the number of correction candidates proposed to the user.

In our approach, the correction of words is done with respect to a regular grammar represented by a finite-state automaton. Thus, we can fully benefit from the optimizations offered by Oflazer's application ([9]) of Du and Chang's approach ([3]). Note that there is no need in [9] for the dictionary to be a finite set of words. It may as well be represented by a regular expression recognizing an infinite set of words.

Consider the following example :

- the dictionary is described by the regular expression $ab^*c + db^*$
- the initial valid word is $A = abc$
- the sequence of updates proposed by the user is $U = (insert(b,3), insert(b,3))$, *i.e.* two insertions of b at the end of the string
- the invalid word resulting from A by the application of U is $B = abcbb$

In the above case the nearest neighbors of B (of distance 2) are : $C_1 = abc$, $C_2 = abbc$, $C_3 = abbbc$ and $C_4 = dbbb$. However, C_2 and C_3 are more plausible correction

[1] Insertions are done *before* the letter on the corresponding position as is the case with [3].

candidates for B than C_1 and C_4 as they seem to better correspond to the user's intention. Proposing C_1 which is equal to the initial word A would ignore the user's wish of modification, while $C_4 = dbbb$ has very little in common with the initial word A that the user is supposed to adapt. Of course, even if C_1 and C_4 are judged less plausible they are never completely discarded as in some cases they may still best suit the user.

The motivation for the incremental string-to-string correction comes from the area of XML-document validation and correction. The validity of each node in such a document is described by a regular expression (in case of a DTD) or by a set of regular expressions (in case of an XML schema). For instance, with [1] the validation is done via a tree automaton whose translation rules are of the form $a, E \to q_a$ where E is a regular expression. Each transition rule indicates that a node having label a and whose children respect the schema rules established by E can be assigned to state q_a. Thus, given a node p labeled with a in the XML tree, a bottom-up automaton performs the validation by verifying whether the word composed by the concatenation of the states (previously) assigned to the children of p belongs to the language $L(E)$.

When a user wishes to modify a valid document but performs a set of invalid updates (*i.e.* leading to an invalid tree) we may start with *locally* validating and correcting the nodes concerned by the updates, together with their closest neighborhood: fathers, siblings, and sons. Since each set of siblings may locally be viewed as a string, we reduce a part of the tree correction to the string-to-string correction problem. Thus, we may often obtain our first valid correction candidates without even touching good parts of the whole tree (those that remain unchanged with respect to the initially valid XML tree) which allows to spare computation time and space and further motivates the notion of incrementality. Our intuition is that such a *shallow correction* approach will often offer the most plausible correction candidates because they vary from the initially valid tree only around the points which the user him/herself wished to modify. At the same time this approach does not exclude a *deep correction* ranging not only over the closest neighbors of the updated nodes but possibly over the whole tree.

The following section describes the computational solution of such incremental string-to-string correction which may be applied locally to an XML-tree on a single-node level.

4 Solution and Algorithms

Let us consider an initially correct word A, *i.e.* A appearing in the language $L(E)$ described by a regular expression E. A user can update A by inserting, deleting or replacing one or more symbols. If the resulting word B happens to be invalid, *i.e.* $B \notin L(E)$, we should propose a set of *valid candidate words*.

We have previously mentioned that in the context of incremental correction the proposed candidate should express the user's intentions as to the modifications of A: it should be obtainable from A by updates similar to those the user him/herself has performed. However, we find it non trivial to define an efficient similarity measure between sequences of updates, which consist of incomparable parameters - operation types, letters, and word positions - and which are non homogeneous (deletions carry no information about letters). Moreover, sequences of updates may show some degree

of redundancy (*e.g.* an operation is performed by one update and later canceled by another update). Therefore, it is not obvious if the user's intentions are best expressed by the updates he wished to perform or by the resulting (invalid) word he/she has produced.

Therefore, we propose an algorithm expressing the similarity of sequences of updates via the similarity of words resulting from these updates. Thus, a valid candidate word is the one that is as close as possible to both A and B, *i.e.* its distance from both A and B doesn't exceed a given threshold. We may calculate the set of such valid candidates applying the Oflazer's ([9]) dynamic programming method to two distance matrices in parallel: the one for the distance between A and C, and the other between B and C. When a particular correction candidate C has been chosen by the user we should instruct him/her on the right updates he/she should have done in order to generate C from A. This right sequence of updates may easily be deduced from the trace between A and C which on its turn may be generated on the basis of the A-C distance matrix.

4.1 Notations

Let E be a regular expression and let $M_E = \langle \Sigma,\ Q,\ \delta,\ q_0,\ F \rangle$ be a deterministic or a non deterministic finite state automaton over an alphabet Σ, a finite set of states Q, an initial state $q_0 \in Q$, a set of accepting states $F \subseteq Q$, and a transition relation $\delta \subseteq Q \times \Sigma \times Q$. Let W be a finite string (or word) of characters (or symbols): $W \in \Sigma^*$. W is valid (correct), iff $W \in L(E)$, where $L(E)$ is a language defined by E. In the following, we introduce some definitions of data types that will be used ahead in this work.

Definition 1. ***Type*** Tr_T ***(a trace)*** *is a list of pairs of integers* $(i,\ j)$ *such that for each* $Tr \in Tr\_T$ *if* $Tr = ((i_1, j_1), (i_2, j_2), \ldots, (i_n, j_n))$ *then*

1. $i_p \neq i_r$ *and* $j_s \neq j_t$ *for* $1 \leq p, r, s, t \leq n$
2. $i_p < i_r$ *iff* $j_p < j_r$ *for* $1 \leq p, r \leq n$

The above definition reformulates the context-independent conditions of the trace definitions by Wagner and Fischer [11] (no character is touched by more than one line, and no two lines cross). In a particular context of two words A and C, a trace $Tr_{A,C} \in Tr\_T$ will always be such that $(Tr_{A,C},\ A,\ C)$ is a minimal cost trace in the sense of [11]. Thus, an extra context-dependent condition, ensuring that lines actually touch character positions of A and B, completes the above definition:

$$Tr_{A,C} = ((i_1, j_1), (i_2, j_2), \ldots, (i_n, j_n)) \text{ where}$$
$$\forall_{1 \leq p \leq n}\ 0 \leq i_p \leq |A| - 1 \text{ and } 0 \leq j_p \leq |B| - 1$$

Recall that there may be several minimal cost traces between A and C.

Definition 2. ***Type*** STr_T ***(a set of traces)*** *describes a set of traces of type* Tr_T *each.*

A particular set of traces $STr_{A,C} \in STr\_T$ will be used in connection with a single pair of words A and C:

$$STr_{A,C} = \{Tr^1_{A,C},\ Tr^2_{A,C},\ \ldots,\ Tr^m_{A,C}\} \text{ where}$$
$$\forall_{i=1,\ldots,m} Tr^i_{A,C} \in Tr\_T \text{and } Tr^i_{A,C} \text{ is a minimal cost trace between } A \text{ and } C.$$

Definition 3. ***Type*** *$SCandid\_T$* ***(a set of candidates):*** *describes a list of elements of the form $(C, (ed_1, ed_2))$, where C is a word, and ed_1, ed_2 are two integers.*

A particular $SCandid_{A,B} \in SCandid\_T$ will be an *ordered* list used in connection with a single pair of words A (valid) and B (invalid), a regular expression E, and a threshold th. $SCandid_{A,B}$ will then describe a set of incremental correction candidates for B with respect to A (see the preceding section).

$SCandid_{A,B} =$
$((C_1, (ed_{A,C_1},\ ed_{B,C_1})), (C_2, (ed_{A,C_2},\ ed_{B,C_2})),\ \ldots, (C_k, (ed_{A,C_k},\ ed_{B,C_k})))$,
where for each $1 \leq i \leq k$, $C_i \in L(E)$, and ed_{A,C_i}, ed_{B,C_i} are the edit distances between A and C_i and between B and C_i respectively.

The ordering of the list is based on the edit distances of the candidates with respect to both A and B. The best correction candidates (found in the front of the $SCandid_{A,B}$ list) are those that are close to both A and B. However, a good candidate may not be equal to A (otherwise the user's intention to modify A would be neglected). For two candidates, if the sums of their distances from A and B are equal then we privilege the candidate that is closer to B (as it's B that best expresses the update intentions of the user). These rules of the ordering of candidates may be formally expressed as follows:

$$
\begin{array}{l}
(C_i, (ed_{A,C_i},\ ed_{B,C_i})) \prec (C_j, (ed_{A,C_j},\ ed_{B,C_j}))\ \textit{iff} \\
(ed_{A,C_j} = 0)\ \textit{or} \\
(ed_{A,C_i} + ed_{B,C_i} < ed_{A,C_j} + ed_{B,C_j})\ \textit{or} \\
(ed_{A,C_i} + ed_{B,C_i} = ed_{A,C_j} + ed_{B,C_j})\ \textit{and}\ (ed_{B,C_i} < ed_{B,C_j})
\end{array}
$$

Definition 4. ***Type*** *H_T* ***(edit distance matrix)*** *is a two dimensional matrix with indices starting from -2. A particular matrix $H_{A,B} \in H\_T$ will always be used in connection with two words A and B so that $H_{A,B}$ is defined as follows:*

$$
\begin{array}{lcll}
H_{A,B}[i,j] & = & H_{A,B}[i-1, j-1], & \textit{if } A[i] = B[j], \\
 & = 1 + & min\{H_{A,B}[i-1, j-1], & \\
 & & \quad H_{A,B}[i-1, j], & \\
 & & \quad H_{A,B}[i, j-1]\} & \textit{otherwise} \\
H_{A,B}[i,-1] & = & i+1, & \textit{for } 0 \leq i \leq |A|-1 \\
H_{A,B}[-1,j] & = & j+1, & \textit{for } 0 \leq j \leq |B|-1 \\
H_{A,B}[-2,j] & = & H_{A,B}[i,-2] = +\infty, & \textit{(boundary definition)}
\end{array}
$$

Note that the above formula is very similar to those used by [9] and [11] for the edit distance calculation. However, there are some minor differences: we do not allow transpositions (contrary to [9]), the cost of each elementary operation is 1 (contrary to [11]), and the numbering of the edit distance matrix indices starts with -2, since the first symbol of a word is indexed by 0.

4.2 Algorithms

Our first algorithm computes all valid candidate words. It contains a recursive procedure, called Explore_rec, that generates new valid words starting with the prefix C,

and whose distance from the given words A and B does not exceed the threshold th. The automaton's state q is the one that has been reached while generating the correction prefix C. New candidates are attached to the list of those found previously ($SCandid$). In its first call, procedure Explore_rec receives, in particular, the initial state q_0, an empty set of candidates $SCandid$, and matrices $H_{A,C}$ and $H_{B,C}$ with their two first columns initialized according to Definition 4 with $C = \epsilon$.

```
1:  procedure Explore_rec (A, B, C, th, H_{A,C},H_{B,C}, M_E, q, SCandid)
2:  input
3:    A: word (a valid word)
4:    B: word (an invalid word resulting from updates of A)
5:    th: integer (error threshold)
6:    M_E: FSA (M_E = ⟨Q, Σ, δ, q_0, F⟩)
7:    q: state (q ∈ Q of M_E, the current state in the automaton)
8:  input/output
9:    C: word (a partial valid candidate word)
10:   H_{A,C}: H_T (edit distance matrix between A and C)
11:   H_{B,C}: H_T (edit distance matrix between B and C)
12:   SCandid: SCandid_T (set of valid candidate words)
13: begin
14:   if (q ∈ F and (H_{A,C}[|A| − 1, |C| − 1] ≤ th) and (H_{B,C}[|B| − 1, |C| − 1] ≤ th))
15:   /** A candidate is found. Candidates are sorted according to Def. 3 **/
16:     SCandid ← SortInsertion(SCandid, (C, (H_{A,C}[|A| − 1, |C| − 1],
17:                                          H_{B,C}[|B| − 1, |C| − 1])))
18:   end if
19:   for each (a, q') ∈ Σ × Q such that δ(q, a) = q'
20:     C ← concat(C, a)
21:     H_{A,C} ← AddNewColumn(H_{A,C}, A, a)
22:     H_{B,C} ← AddNewColumn(H_{B,C}, B, a)
23:     if ((cuted(A, C, H_{A,C}, th) ≤ th) and (cuted(B, C, H_{B,C}, th) ≤ th))
24:       Explore_rec(A, B, C, th, H_{A,C}, H_{B,C}, M_E, q', SCandid)
25:     end if
26:     H_{A,C} ← DeleteLastColumn(H_{A,C})
27:     H_{B,C} ← DeleteLastColumn(H_{B,C})
28:     C ← DelLastSymbol(C)
29:   end for each
30: end
```

The automaton M_E is explored in the depth-first order. Each time a transition is followed the current prefix C is extended (line 20) and new columns are added to both distance matrices (lines 21–22). That allows to check if C may still lead to a candidate remaining within the distance threshold from A and B (line 23). If it does the path is followed via a recursive call (line 24), otherwise the path gets cut off. In each case the transition is finally backed off (lines 26-28) and a new transition outgoing from the same state is tried out. If we arrive at a final state and the distance from C to both A and B does not exceed the threshold (line 14) then C is a valid candidate that gets inserted to the list of all candidates found so far (lines 16–17). The insertion is done according to Definition 3.

Note that the validation of the extended C with respect to the threshold (line 23) is done via the function *cuted* that computes the cut-off edit distance between A and

C, and between B and C, as defined by [9]. It corresponds to the minimum value of the current column in the edit distance matrix (*i.e.* the column corresponding to the last character in the extended C). It has been shown by [9] that if this value exceeds the threshold then there is no chance for further columns not to exceed the threshold. Thus, C may not be a prefix of a valid word whose distance from A and B is lower than the threshold.

Let's consider, for instance, a grammar $E = (aba+bab)^*$ and a valid word $A = bab$. If we apply the sequence of updates $S = (insert(a, 1), replace(a, 2))$ to A we obtain an invalid word $B = baaa$. For $th = 2$ the above function returns the following list of candidates: $SCandid = ((aba, (2, 2)),\ (bab, (0, 2)))$.

Given an ordered list of correction candidates the user may choose the one that best fits his/her needs. However, we also wish to show the user how to obtain C from A in order to let him/her avoid the same errors in future. The sequence of updates needed to take A to C can easily be deduced from a minimum cost trace between these two words. In the following we present a recursive function Trace_rec, that allows the construction of all minimal cost traces transforming A into C.

```
1:  function Traces_rec (A, C, H_{A,C}, i, j, Tr)
2:  input
3:    A: word (a valid word before updates)
4:    C: word (a valid candidate word)
5:    H_{A,C}: matrix (edit distance matrix between A and C)
6:    i, j: integers (indices of the current element of H_{A,C})
7:    Tr: Tr_T (a partial trace between A and C)
8:  result: STr_T (a set of traces between A and C)
9:  local variable
10:   STr: STr_T (a set of partial traces between A and C)
11: begin
12:   STr ← ∅      /* initialization */
13:   if ((i ≠ −1) or (j ≠ −1))
14:     if (H_{A,C}[i, j] = H_{A,C}[i − 1, j] + 1)      /* deletion */
15:       STr = STr ∪ Traces_rec(A,C,H_{A,C},i − 1,j,Tr) end if
16:     if (H_{A,C}[i, j] = H_{A,C}[i, j − 1] + 1)      /* insertion */
17:       STr = STr ∪ Traces_rec(A,C,H_{A,C},i,j − 1,Tr) end if
18:     if ((H_{A,C}[i, j] = H_{A,C}[i − 1, j − 1] + 1) and (A[i] ≠ C[j])) /*replacement*/
19:       STr = STr ∪ Traces_rec(A,C,H_{A,C},i − 1,j − 1,HeadInsert(Tr, (i, j))
20:       end if
21:     if ((H_{A,C}[i, j] = H_{A,C}[i − 1, j − 1]) and (A[i] = C[j])) /*no operation*/
22:       STr = STr ∪ Traces_rec(A, C, H_{A,C}, i − 1, j − 1, HeadInsert(Tr, (i, j))
23:       end if
24:   else
25:     STr = STr ∪ {Tr}
26:   end if
27:   return(STr)
28: end
```

The function runs over the error distance matrix from its bottom right-hand corner to its top left-hand corner. For the current matrix' element (i, j) the last parameter Tr holds all partial traces allowing to transform $A[i : |A| - 1]$ to $C[j : |A| - 1]$. In its first call the function receives an empty set of partial traces Tr, as well as $i = |A| - 1$

and $j = |C| - 1$, the indices of the bottom right-hand element of the matrix, *i.e.* the one that contains the edit distance between A and C.

In order to find a minimum cost trace between A and B it is sufficient to recall how the relevant elements of the error distance matrix $H_{A,C}$ have been calculated. The relevant elements are those that directly contribute to the computation of the final bottom left-hand element of $H_{A,C}$. Recall that each element $H_{A,C}[i,j]$ has been deduced in the procedure Explore_rec by the *AddNewColumn* function from one of its three top left-hand neighboring elements:

1. If $H_{A,C}[i,j]$ is equal to $H_{A,C}[i-1,j]+1$ it means that $C[0:j]$ can be obtained from $A[0:i]$ by the same edit operations as those needed for transforming $A[0:i-1]$ to $C[0:j]$, and by an additional deletion of $A[i]$ at position i. Thus, the trace between $A[0:i]$ and $C[0:j]$ is the same as the trace between $A[0:i-1]$ and $C[0:j]$ (line 15) since the letters to be deleted don't appear in the trace.
2. If $H_{A,C}[i,j]$ is equal to $H_{A,C}[i,j-1]+1$ it means that $C[0:j]$ can be obtained from $A[0:i]$ by the same edit operations as those needed for transforming $A[0:i]$ to $C[0:j-1]$, and by an additional insertion of $C[j]$ at position $i+1$ (line 17) as insertions occur before the given position. The trace between $A[0:i]$ and $C[0:j]$ is the same as between $A[0:i]$ and $C[0:j-1]$ since the letters to be inserted don't appear in the trace.
3. If $H_{A,C}[i,j]$ is equal to $H_{A,C}[i-1,j-1]+1$ and $A[i]$ is different from $C[j]$ it means that $C[0:j]$ can be obtained from $A[0:i]$ by the same edit operations as those needed for transforming $A[0:i-1]$ to $C[0:j-1]$, and by an additional replacement of $A[i]$ by $C[j]$ at position i. Thus, the trace between $A[0:i]$ and $C[0:j]$ is the same as the trace between $A[0:i-1]$ and $C[0:j-1]$ to which a replacement line of $A[i]$ by $C[j]$ has been added (line 19).
4. If $H_{A,C}[i,j]$ is equal to $H_{A,C}[i-1,j-1]$ and $A[i]$ is equal to $C[j]$ it means that $C[0:j]$ can be obtained from $A[0:i]$ by the same edit operations as those needed for transforming $A[0:i-1]$ to $C[0:j-1]$. Thus, the trace between $A[0:i]$ and $C[0:j]$ is the same as the trace between $A[0:i-1]$ and $C[0:j-1]$ to which an identity line between $A[i]$ and $C[j]$ has been added (line 21).

Let's consider the same example as on page 211. For candidate *aba* the above function returns the following set of minimum cost traces: $\{((0,1),(1,2)),\ ((1,0),(2,1))\}$.

5 Complexity and Experimental Results

Let $n = min(|A|,|B|)$ where B is the invalid word to be corrected, resulting from a valid word A. Let f_{max} be the maximum fan-out of our automaton M_E. Procedure Explore_rec has to perform, at worst, a depth-first exploration of M_E in which the depth of each path comes up to $n+th$ (because a word staying within the threshold th from both A and B may not be longer than $n+th$). Thus, the worst-case complexity of this procedure is $O(f_{max}^{n+th})$.

Function Traces_rec is called after procedure Explore_rec has determined the list of all candidates. At that moment the $H_{A,C}$ matrix for a candidate C chosen by the user does not exist any more and has to be recalculated which takes a time proportional to $|A| * |C|$. Function Traces_rec has to cross the error distance matrix from the bottom

Regular expression	Threshold	Number of updates	Number of candidates	Execution time(ms)
$E = (a\|b)c(d\|e)$	0	0	1	1
	1	2	1	1
	2	3	2	10
	3	4	1	1
	5	3	4	10
$E' = (a\|b)^*c(d\|e?)$	0	0	1	1
	1	2	3	1
	2	3	17	10
	3	4	10	1
	5	3	117	40

Table 1: Number of candidates and execution time obtained for the initial word *acd* when dealing with starred and non-starred regular expressions.

right-hand to the top left-hand corner in order to find all traces corresponding to the given candidate. In each position the path may only continue west, north or north-west. Since the matrix's size is no bigger than $n \times (n+th)$, the number of all possible recursive calls is less than $\sum_{i=1}^{n+th} 3^i = 3/2 * (3^{n+th-1} - 1)$. So the complexity of the trace calculation is $O(n^2 + 3^{n+th}) = O(3^{n+th})$.

Hence, the worst-case complexity of finding all candidates, and all traces for one chosen candidate is $O(f_{max}^{n+th}) + O(3^{n+th}) = O(c^{n+th})$ where $c = max(f_{max}, 3)$.

Although the complexity of our method seems to be discouraging, the worst cases rarely happen in practice. Our experimental results show that our algorithm is fast and gives good results in most cases. Our implementation was done in Java (JRE 1.4.1) running under Windows 2000. We use a 800 MHz Celeron Pentium system with 392 Mbytes of memory and a 40 GB hard disk with 5400 rpm.

We have performed 160 experiments by varying the regular expression, the threshold, the size of the initial word, and the number of updates. The statistical measures, chosen among those that are not disproportionately affected by extreme scores ([5]), give the following results: the *median* (the value separating the highest half from the lowest half of the results) is equal to 10 ms, the *mode* (the most frequent result) is 1 ms, and mean execution time of the 90% fastest runs is 44 ms.

We further examined the importance of different parameters on the number of candidates proposed by the program, and on its execution time. We noticed that the existence of starred sub-expressions, possibly embedded (*e.g.* $((ab)^*c)^*$) or ranging over a disjunction (*e.g.* $(a|b)^*c^*$), has a crucial importance for these two results.

Table 1 presents two test sets corresponding to regular expressions with and without Kleene-operators. In each test set, we varied two parameters: the error threshold and the number of updates. Columns 4 and 5 give the number of candidates generated by our method, together with the time needed for this computation.

We notice that for the same word a starred expression allows more correction candidates and their computation time may be several times higher than in the case of a non-starred expression. The reason is that the algorithm tries to compose different words containing repetitive characters within the range of the starred part of the

Label	Candidate	edit_distance(A, C_i)	edit_distance(B, C_i)
C_1	aaaaaacd	2	1
C_2	aaaaacd	1	2
C_3	aaaaabcd	2	2
C_4	aaaabacd	2	2
C_5	aaabaacd	2	2
C_6	aabaaacd	2	2
C_7	abaaaacd	2	2
C_8	baaaaacd	2	2

Table 2: Candidates for $E' = (a|b)^*c(d|e?)$, $A = aaaacd$ (valid), $B = aaaaacacd$ (invalid), and $th = 2$.

expression. For instance, given the regular expression $E' = (a|b)^*c(d|e?)$, the initial valid word $A = aaaacd$, the resulting invalid word $B = aaaaacacd$, and threshold 2, all correction candidates are obtained by modifying the subsequence recognizable by the subexpression $(a|b)^*$ while the suffix, recognizable by $c(d|e?)$, remains intact (see Table 2).

Our intuition is that word subsequences corresponding to starred sub-expressions, such as $(a|b)^*$, could be treated as blocks, so that their modification is not proposed if none of the user's updates falls within the range of the starred sub-expression. This heuristic might allow some optimizations of our method.

6 Conclusions and Future Work

We have introduced the problem of an incremental string-to-string correction: given a regular grammar E, a valid word A and a sequence S of updates (insertions, deletions, and replacements of letters) that transform A into an invalid word B, find all valid words C that may result from A by sequences of updates that are as similar as possible to S.

It seems non trivial to define an efficient similarity measure between sequences of updates. Therefore, we proposed an algorithm that addresses the above problem by expressing the similarity of sequences of updates via the similarity of words resulting from these updates. Thus, an incremental string correction may be implemented by the nearest-neighbor search in a finite-state automaton performed simultaneously for both A and B within a given threshold, according to algorithms proposed by [11], [3] and [9]. The reconstruction of a *trace* between the initial valid word and a correction candidate chosen by the user allows him/her to know the right update sequence needed to obtain this candidate.

Despite an exponential worst-case complexity (frequent in approaches based on an extensive finite-state automaton exploration), our algorithm gives good experimental results calculated over a large sample of tests with varying parameters. We think that some optimizations, concerning both the candidate's pertinence and the execution time, may be done if the internal structure of the regular expression is taken into account, particularly with respect to Kleene's operators. Moreover, it is also possible

to examine optimizations resulting from recent approaches to approximate search in a dictionary such as [8].

Another factor worth examination is the possibility of admitting two different threshold values for the two words A and B. That seems particularly relevant in the case of long sequences of updates: if the threshold is much lower than the number of updates the user wished to perform then there is a small chance for a candidate remaining within this threshold distance from A to reflect the user's intentions. For example, if the user has performed 10 updates he/she will probably not be satisfied with candidates the vary only by one or two operations from the initial word A. Admitting a higher threshold with respect to A and the lowest possible threshold with respect to B seems a good strategy that we wish to experiment on.

The definition of an incremental string-to-string correction problem is inspired by the domain of incremental XML-document correction, in which an initially valid XML-tree is taken into account in order to limit the correction space to contexts surrounding the points of updates. Thus, naturally, our main perspective is the extension of the presented method to deeper tree structures in which not only a node's siblings but possibly also its ancestors and descendants are taken into account.

References

[1] B. Bouchou and M. Halfeld Ferrari Alves. Updates and Incremental Validation of XML Documents. In *9th International Workshop on Data Base Programming Languages (DBPL), Potsdam, Germany*, 2003.

[2] G. Clarke D. T. Barnard and N. Duncan. Tree-to-tree Correction for Document Trees. Technical Report 95-372, Department of Computing and Information Science, Queen's University, Kingston, Ontario, 1995.

[3] M. W. Du. and S. C. Chang. A model and a fast algorithm for multiple errors spelling correction. *Acta Informatica*, 29:281–302, 1992.

[4] P. A. V. Hall and G. R. Dowling. Approximate String Matching. *Computing Surveys*, 12(4):381–402, 1980.

[5] David C. Howell. *Fundamental Statistics for the Behavioral Sciences*. Library of Congress Cataloging-in-Publication Data, 4th ed., 1999.

[6] K. Kukich. Techniques for Automatically Correcting Words in Text. *ACM Computing Surveys*, 24(4):377–439, 1992.

[7] R. Lowrance and R. A. Wagner. An Extension of the String-to-String Correction Problem. *Journal of the ACM*, 22(2):177–183, 1975.

[8] S. Mihov and K. U. Schulz. Fast approximate search in large dictionaries. *Computational Linguistics*, 30(4):451–477, 2004.

[9] K. Oflazer. Error-tolerant Finite-state Recognition with Applications to Morphological Analysis and Spelling Correction. *Computational Linguistics*, 22(1):73–89, 1996.

[10] R. A. Wagner. Order-n Correction for Regular Languages. *Communications of the ACM*, 17(5):265–268, 1974.

[11] R. A. Wagner and M. J. Fischer. The String-to-String Correction Problem. *Journal of the ACM*, 21(1):168–173, 1974.

A Missing Link in Root-to-Frontier Tree Pattern Matching

Loek G. W. A. Cleophas, Kees Hemerik and Gerard Zwaan

Department of Mathematics and Computer Science,
Technische Universiteit Eindhoven,
P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands

e-mail: loek@loekcleophas.com, c.hemerik@tue.nl, g.zwaan@tue.nl

Abstract. Tree pattern matching (TPM) algorithms play an important role in practical applications such as compilers and XML document validation. Many TPM algorithms based on tree automata have appeared in the literature. For reasons of efficiency, these automata are preferably deterministic. Deterministic root-to-frontier tree automata (DRFTAs) are less powerful than nondeterministic ones, and no root-to-frontier TPM algorithm using DRFTAs has appeared so far. Hoffmann & O'Donnell [HO82] presented a root-to-frontier TPM algorithm based on an Aho-Corasick automaton recognizing tree stringpaths, but no relationship between this algorithm and algorithms using tree automata has been described. In this paper, we show that a specific DRFTA can be used for stringpath matching in a root-to-frontier TPM algorithm. This algorithm has not appeared in the literature before, and provides a missing link between TPM algorithms using stringpath automata and those using tree automata.

1 Introduction

Tree pattern matching (TPM) is an important problem from regular tree theory. It can be described as finding all occurrences of one or more given pattern trees (patterns) in a given subject tree. Algorithms solving this problem form the basis for tree acceptance and tree parsing algorithms, which play an important role in practical applications such as compilers and XML document validation.

The problems of pattern matching, acceptance and parsing for trees are related, and so are the algorithms solving them (referred to as *tree algorithms* from this point onward). Either implicitly or explicitly, many use some form of tree or string automata, combined with a frontier-to-root (bottom-up) or root-to-frontier (top-down) tree traversal [FSW94, HK89, AGT89, vdM88, vD87, Cha87, HC86, AG85, HO82, Kro75]. For efficiency reasons, the automata used should preferably be deterministic.

In frontier-to-root tree algorithms, deterministic frontier-to-root tree automata can be and indeed often are used [FSW94, HK89, Cha87, HC86, HO82].

Deterministic root-to-frontier tree automata (DRFTAs) however have not been used in root-to-frontier tree algorithms, since it is known from regular tree theory that in general they are less powerful than their nondeterministic counterparts (NRFTAs) [Eng75, GS97]. Such algorithms therefore use NRFTAs [vD87].

Hoffmann & O'Donnell [HO82] and Aho, Ganapathi & Tjiang [AGT89, AG85] presented root-to-frontier tree algorithms based on deterministic *string* automata instead. These algorithms use a deterministic Aho-Corasick (AC) automaton [AC75] and its output function to recognize tree *stringpaths*. Since a tree is uniquely determined by its stringpaths, this automaton can be used to detect tree matches. The presentation in those papers is somewhat informal and complicated by optimizations, but van de Meerakker [vdM88] gave a stepwise account of how to obtain the algorithms. Unfortunately, no relationship between tree algorithms using stringpath automata and those using tree automata seems to have appeared in the literature.

In this paper, we show that even though a DRFTA cannot be used as a tree acceptor or matcher by itself, a specific DRFTA can be used (together with an output function) for stringpath matching in a root-to-frontier tree traversal. We present a version of Hoffmann & O'Donnell's root-to-frontier TPM algorithm that uses an AC automaton and output function, and then present a modified TPM algorithm that uses this DRFTA and associated output function. To the best of our knowledge, this algorithm has not appeared in the literature before. It provides a missing link between TPM algorithms using stringpath automata and those using tree automata.

Our algorithm is not necessarily efficient. It has the same worst-case bound of $\mathcal{O}(m \cdot n)$ as the other papers mentioned (where m and n are the pattern and subject tree size). In recent years, many papers providing algorithms with better worst-case bounds have appeared [Kos89, DGM94, CH97, CHI99]. These algorithms improve the worst-case bound at the cost of somewhat more elaborate algorithms and/or the construction of larger auxiliary data structures. It is unclear (and a potential subject of future research) what the practical performance of the algorithms is.

Section 2 introduces basic definitions and notations. Tree pattern matching is introduced in Section 3. In Section 4 we present a version of Hoffmann & O'Donnell's root-to-frontier TPM algorithm, while Section 5 shows that a particular kind of DRFTA can be constructed and used for stringpath matching, and presents a root-to-frontier TPM algorithm using this DRFTA. Section 6 gives some conclusions as well as suggestions for future work, in particular a more detailed comparison of the two algorithms and automata kinds.

2 Preliminaries

We use $\mathbb{B}, \mathbb{N}$ and $\mathbb{N}_+$ to denote the domain of the booleans, the natural numbers and the natural numbers excluding 0 respectively.

A basic understanding of the meaning of *quantifications* is assumed. We use the notation $\langle \oplus a : R(a) : E(a) \rangle$ where $\oplus$ is the associative and commutative *quantification operator* (with unit $e_\oplus$), a is the *quantified variable* introduced, R is the *range predicate* on a, and E is the *quantified expression*. By definition, we have $\langle \oplus a : false : E(a) \rangle = e_\oplus$. The following table lists some of the most commonly quantified operators, their quantified symbols, and their units:

Operator	$\vee$	$\wedge$	$\cup$
Symbol	$\exists$	$\forall$	$\bigcup$
Unit	$false$	$true$	$\varnothing$

We use $\langle \mathbf{Set}\, a :\ R(a)\ :\ E(a)\ \rangle$ for $\langle \bigcup a :\ R(a)\ :\ \{E(a)\}\ \rangle$.

2.1 Trees

Definition 1. An *ordered tree domain* is a finite non-empty subset D of $\mathbb{N}_+^*$ such that

- $\mathbf{pref}(D) \subseteq D$, i.e. D is prefix-closed, and
- for all $\mathfrak{n} \in D$ and $i \in \mathbb{N}_+$, $\mathfrak{n} \cdot i \in D \Rightarrow \langle \forall\, j :\ j < i\ :\ \mathfrak{n} \cdot j \in D \rangle$.

Note that we use $\cdot$ to separate elements of $\mathbb{N}_+$ in tree domain elements. Tree domain elements are called *nodes*. *Root node* $\varepsilon \in D$ for any D. □

Example 2. Set $\{\varepsilon, 1, 1 \cdot 1, 2\}$ is an ordered tree domain. □

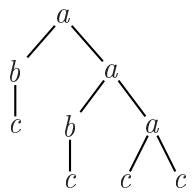
Definition 3. Let D be an ordered tree domain, V a finite non-empty set of symbols (alphabet) and $r \in V \to \mathbb{N}$ a *ranking function*. An *ordered ranked tree* t is a function $t \in D \to V$ for which for every node $\mathfrak{n} \in D$, $r(t(\mathfrak{n}))$ equals the number of $i \in \mathbb{N}_+$ such that $\mathfrak{n} \cdot i \in D$. □

For a tree t, we will use D_t to refer to the tree domain underlying t. Given an alphabet V and ranking function r, we call the pair (V, r) a ranked alphabet. For any $a \in V$, we call $r(a)$ the rank of a. In this paper, we assume (V, r) to be the fixed ranked alphabet $\{(a, 2), (b, 1), (c, 0)\}$, i.e. with symbols a, b, c of rank $2, 1, 0$.

We denote the set of all ordered ranked trees over (V, r) by $\mathit{Tree}(V, r)$. For $t \in \mathit{Tree}(V, r)$ and $\mathfrak{n} \in D_t$, $t@\mathfrak{n}$ is t's subtree starting at $\mathfrak{n}$. Note that $t@\varepsilon = t$.

We may represent a tree by a set of pairs of tree domain values and symbols, graphically, or using a term representation, as in the following example.

Example 4 (Tree). Given (V, r), set $t = \{(\varepsilon, a), (1, b), (1 \cdot 1, c), (2, a), (2 \cdot 1, b), (2 \cdot 1 \cdot 1, c), (2 \cdot 2, a), (2 \cdot 2 \cdot 1, c), (2 \cdot 2 \cdot 2, c)\}$ forms an ordered, ranked tree. It can be represented as a term by $a(b(c), a(b(c), a(c, c)))$, while tree $t@(2 \cdot 2)$ for example corresponds to set $\{(\varepsilon, a), (1, c), (2, c)\}$ and term $a(c, c)$. Graphically, t is represented as



□

2.2 Tree automata

Definition 5. A *tree automaton* (TA) M is a 6-tuple $(Q, V, r, R, Q_{ra}, Q_{la})$ such that

- Q is a finite set, the *state set*
- (V, r) is a ranked alphabet
- $R = \langle \mathbf{Set}\, a :\ a \in V\ :\ R_a \rangle$ is the set of transition relations, where $R_a \subseteq Q \times Q^{r(a)}$ for all $a \in V$
- $Q_{ra} \subseteq Q$, the *root accepting states*

- $Q_{la} \subseteq Q$, the *leaf accepting states*,
  defined by $Q_{la} = \langle \mathbf{Set}\, a, q : \ a \in V \wedge r(a) = 0 \wedge (q, ()) \in R_a \ : \ q \ \rangle$

□

Remark 6. An explicit set Q_{la} for *leaf accepting states* is not needed, but is included to facilitate notation. Note that for $a \in V$ with $r(a) = 0$, $R_a \subseteq Q \times Q^0$, i.e. the second component corresponds to a domain whose single element is the empty tuple $()$. Some definitions of tree automata use $R_a \subseteq Q \times Q$ for such symbols a instead. □

Definition 7. An NRFTA (nondeterministic root-to-frontier tree automaton) $M = (Q, V, r, R, Q_{ra}, Q_{la})$ is a TA where $R_a \in Q \to \mathcal{P}(Q^{r(a)})$ for all $a \in V$, i.e. R_a is considered to be directed. □

Considering the relations R_a in this way is not a restriction, and therefore the classes of NRFTA and TA are equivalent. By directing the relations, the root accepting states become start states. By restricting the relations R_a of the NRFTA to be functions yielding a single state tuple instead of a set of such tuples, we obtain the deterministic root-to-frontier tree automata:

Definition 8. A DRFTA $M = (Q, V, r, R, Q_{ra}, Q_{la})$ is an NRFTA where $R_a \in Q \to Q^{r(a)}$ for all $a \in V$—i.e. the R_a are functions—and $Q_{ra} = \{q_{ra}\}$—i.e. there is a unique root accepting state (start state). □

We define *tree acceptance* using tree state assignments, i.e. assignments of a state to each tree node. Consider the set of tree state assignments that respect the automaton transition relations (or functions in case of directed automata) and that assign a (the, for DRFTAs) root accepting state to the subject tree root. A subject tree is accepted by an automaton if and only if this set is non-empty.

Lemma 9. There are NRFTAs for which no DRFTA accepting the same language can be constructed.

Proof. We give an example of a language which is not recognizable by a DRFTA.

Let $L = \{a(c, d), a(d, c)\}$. We try to construct a DRFTA accepting L. There must be exactly one pair of states (q_1, q_2) such that $R_a(q_{ra}) = (q_1, q_2)$. To recognize both trees, $R_c(q_1) = R_d(q_2) = R_d(q_1) = R_c(q_2) = ()$ must hold, but this means that trees $a(c, c)$ and $a(d, d)$ are accepted as well. A DRFTA accepting L therefore cannot exist, but an NRFTA for L can be constructed (see Figure 1, the notation of which is explained below). □

Finite string automata are often represented visually by a state diagram. We adapt this notation to finite tree automata. Each state is represented by a circle, with double circles indicating root accepting states, while a transition relating state q and states $q_1 \ldots q_n$ on a symbol a is represented by

1. a (directed) edge connecting q to a small unlabeled circle, labeled by a and
2. n (directed) edges connecting the unlabeled circle to q_i (for $1 \leq i \leq n$)

Finally, we introduce dotted trees, which are used in Section 5. A dotted tree is a tree with a distinguished position, as in the following definition.

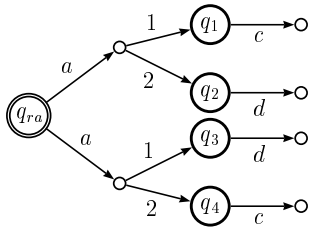


Figure 1: NRFTA accepting $L = \{a(c,d), a(d,c)\}$

Definition 10. Let $t \in \mathit{Tree}(V,r)$ and $\mathfrak{n} \in D_t$, then the pair $(t,\mathfrak{n})$ is a *dotted tree*. We use $DT(t)$ to indicate the set of all dotted trees for a tree t. □

Example 11. Let $u = a(b(c),c)$, then set $DT(u)$ corresponds to $\{(u,\varepsilon),(u,1),(u,1\cdot 1),(u,2)\}$. □

3 Tree pattern matching

The leaves of trees in $\mathit{Tree}(V,r)$ always have symbols of rank 0, but for pattern matching, something more general is needed. We extend the alphabet with a special variable or 'wildcard' symbol, indicating a match of any tree from $\mathit{Tree}(V,r)$. We extend (V,r) into (V',r') by adding symbol ν with $r'(\nu) = 0$, and letting $r'(a) = r(a)$ for all $a \in V$. Trees in $\mathit{Tree}(V',r')$ are called *pattern trees* or *patterns*. Note that ν can only label leaf nodes. Notation $t@\mathfrak{n}$ and $DT(t)$ are extended to trees in $\mathit{Tree}(V',r')$.

We can now define what it means for a subtree of a tree to match a pattern, defining a function *Match* as follows.

Definition 12. Function $\mathit{Match} \in \mathit{Tree}(V',r') \times \mathit{Tree}(V,r) \times D \to \mathbb{B}$ is defined for every pattern $p \in \mathit{Tree}(V',r')$, subject $t \in \mathit{Tree}(V,r)$ and node $\mathfrak{n} \in D_t$ by $\mathit{Match}(p,t,\mathfrak{n}) =$

$$\left\langle \exists\, s_1,\ldots,s_k :\ s_1,\ldots,s_k \in \mathit{Tree}(V,r)\ :\ p[s_1,\ldots,s_k] = t@\mathfrak{n} \right\rangle$$

where $p[s_1,\ldots,s_k]$ is the tree obtained by substituting $s_1,\ldots,s_k$ respectively for the k instances of ν in p. □

Example 13. Given trees $t = a(b(c),a(b(c),a(c,c)))$ and $p = a(b(c),\nu)$, $\mathit{Match}(p,t,\mathfrak{n})$ holds for $\mathfrak{n} = \varepsilon$ and $\mathfrak{n} = 2$ (and not for any other nodes). $\mathit{Match}(p,t,\varepsilon)$ holds since $t@\varepsilon = p[a(b(c),a(c,c))]$. $\mathit{Match}(p,t,2)$ holds since $t@2 = p[a(c,c)]$. □

Apart from the tree domain, term and graphical notations used before, a tree is also uniquely characterized by its set of stringpaths, which represent all its root to leaf paths.

Definition 14 (Tree stringpaths). Let $t \in \mathit{Tree}(V',r')$, then function $\mathit{SPaths} \in \mathit{Tree}(V',r') \to \mathcal{P}((V' \cdot \mathbb{N}_+)^* \cdot V')$ is defined by

$$\begin{aligned} \mathit{SPaths}(t) &= \{t(\varepsilon)\} && \text{if } r(t(\varepsilon)) = 0 \\ \mathit{SPaths}(t) &= \{t(\varepsilon)\} \\ &\quad \cdot \left\langle \bigcup i :\ 1 \le i \le r(t(\varepsilon))\ :\ \{i\} \cdot \mathit{SPaths}(t@i) \right\rangle && \text{if } r(t(\varepsilon)) > 0 \end{aligned}$$

(where string concatenation operator $\cdot$ is extended to operate on sets of strings). □

Example 15. For $t = a(b(c), a(b(c), a(c, c)))$, $SPaths(t@2) = \{a1b1c, a2a1c, a2a2c\}$ and $SPaths(t) = \{a1b1c,\ a2a1b1c, a2a2a1c, a2a2a2c\}$. □

A stringpath of a pattern p matches in a given subject tree t starting at $\mathfrak{n}$ if and only if either the stringpath is in $SPaths(t@\mathfrak{n})$ or the stringpath ends in ν and the stringpath minus this ν is a prefix of some stringpath in $SPaths(t@\mathfrak{n})$. It follows that p matches in t at node $\mathfrak{n}$ if and only if each stringpath in $SPaths(p)$ matches in t starting at $\mathfrak{n}$.

We introduce infix operators $\restriction$ and $\downharpoonright$ (*right take* and *right drop*). For any string s of length $\geq m \in \mathbb{N}_+$, $s \restriction m$ equals the rightmost m symbols of s, while $s \downharpoonright m$ equals s except its rightmost m symbols.

Example 16. Given tree $t = a(b(c), a(b(c), a(c, c)))$ and pattern $p = a(b(c), \nu)$, $Match(p, t, \mathfrak{n})$ holds for $\mathfrak{n} = \varepsilon$ and $\mathfrak{n} = 2$ only. $Match(p, t, \varepsilon)$ holds since $a1b1c \in SPaths(t@\varepsilon)$ and $a2\nu\restriction 1 = \nu \wedge a2\nu\downharpoonright 1 \in \mathbf{pref}(SPaths(t@\varepsilon))$. $Match(p, t, 2)$ holds since $a1b1c \in SPaths(t@2)$ and $a2\nu\restriction 1 = \nu \wedge a2\nu\downharpoonright 1 \in \mathbf{pref}(SPaths(t@2))$. □

To solve the TPM problem using a root-to-frontier approach, stringpath matching can be used. Stringpath matches are most easily registered at their endpoints, but algorithms can be adapted to register stringpath matches at their beginpoints, and by doing so, tree pattern matches can be determined. In the rest of this paper, we consider tree pattern matching as stringpath matching.

Related to the definition of stringpaths, we define a function representing the rootpath to a given node, i.e. the labeled path from the tree root to the given node:

Definition 17. Function $RPath \in Tree(V', r') \times D \to (V' \cdot \mathbb{N}_+)^* \cdot V'$ is defined by

$$\begin{aligned} RPath(t, \varepsilon) &= t(\varepsilon) \\ RPath(t, \mathfrak{n} \cdot i) &= RPath(t, \mathfrak{n}) \cdot i \cdot t(\mathfrak{n} \cdot i) \text{ for } \mathfrak{n} \cdot i \in D_t \end{aligned}$$

□

Note that a rootpath $RPath(t, \mathfrak{n})$ always ends with symbol $t(\mathfrak{n})$.

For every pattern p, there is a correspondence between dotted trees and rootpaths: $RPath(p, \mathfrak{n})$ is defined if and only if $(p, \mathfrak{n}) \in DT(p)$.

4 Using AC stringpath automata

The basic idea of Hoffmann & O'Donnell's root-to-frontier TPM algorithm [HO82] is to use an optimal AC automaton for matching pattern stringpaths, combined with a root-to-frontier traversal of the subject tree.

An optimal AC automaton is a version of the AC automaton without failure transitions. Construction algorithms for AC automata have been described in numerous references [CR03, NR02, Wat95, AC75], and we do not discuss any in detail.

Given the state reached by the AC automaton by processing an input string upto a given position, the output function determines the set of keyword occurrences ending at this position.

The AC automaton built from stringpath set $SPaths(p)$ for a given pattern $p \in Tree(V', r')$ is a 5-tuple $M_{AC} = (Q, V' \cup \mathbb{N}_+, \delta, q_0, output)$ in which Q is the state set,

$V' \cup \mathbb{N}_+$ the alphabet, $\delta \in Q \times (V' \cup \mathbb{N}_+) \to Q$ the transition function, q_0 the start state, and $output \in Q \to \mathcal{P}(SPaths(p))$ the output function.

On a high level, the construction of this automaton can be described as follows:

1. Construct a *trie* recognizing the set of stringpaths
2. For every state corresponding to a stringpath match, define the output of the state equal to the stringpath; for other states, the output is empty
3. Add a 'self-loop' transition on every alphabet symbol to the start state
4. Determinize the resulting automaton and adapt the output function accordingly

The resulting optimal AC automaton for the set of pattern stringpaths can be used in a root-to-frontier subject tree traversal to find all pattern stringpath matches.

Example 18 (AC stringpath automaton for pattern p). The trie with 'self-loop' constructed for pattern $p = a(b(c), \nu)$ by steps 1–3 of the above construction is depicted in Figure 2. The output function values corresponding to final states are defined as $output(q_c) = a1b1c, output(q_d) = a2\nu$.

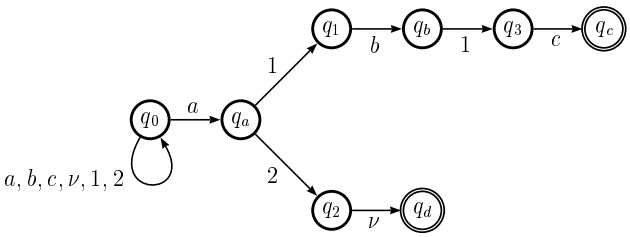


Figure 2: Trie with 'self-loop' for p

Applying step 4 of the above construction leads to the AC stringpath automaton depicted in Figure 3 (in which transitions not shown lead to q_0). The output function values corresponding to final states are defined as $output(q_c) = \{a1b1c\}, output(q_d) = \{a2\nu\}$. Note that states in the AC automaton are different from those in the trie with 'self-loop', since states of the AC automaton correspond to sets of states of the trie with 'self-loop'. □

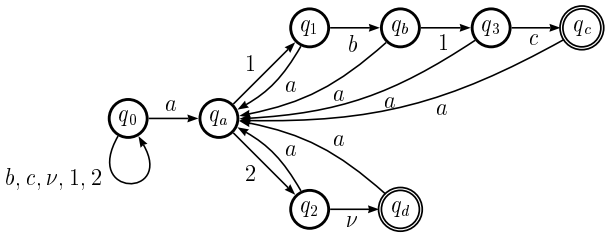


Figure 3: AC stringpath automaton for p.

4.1 An AC-based TPM algorithm

In this section, we present a version of Hoffmann & O'Donnell's root-to-frontier TPM algorithm. The algorithm presentation is similar to that by van de Meerakker [vdM88]. It uses explicit recursion instead of a stack as in the original algorithm. As an invariant, when visiting a node $\mathfrak{n}$ of the given subject tree t, the AC

automaton is in the state reached on input equal to the rootpath $RPath(t, \mathfrak{n})$ except its last symbol, $t(\mathfrak{n})$, i.e. on input $RPath(t, \mathfrak{n}) \lfloor 1$.

To traverse the tree, the algorithm should be called on every child node i of the current node, if any. To maintain the invariant, the AC automaton should be in the state reached from the current state by a transition on $t(\mathfrak{n})$ followed by one on i, the number of the branch leading to the child node.

When visiting a node, the algorithm should register matches indicated by the AC automaton after a transition on symbol $t(\mathfrak{n})$, but also matches indicated after a transition on symbol ν, since ν matches any subtree. This results in:

{ Pre: $q = \delta^*(q_0, RPath(t, \mathfrak{n}) \lfloor 1)$ }
proc $Traverse(q : Q, \mathfrak{n} : D) =$
[[**var** $q_{next} : Q$; $i : \mathbb{N}_+$; $sp : (V \cdot \mathbb{N}_+)^* \cdot V$
| **for** $i : 1 \leq i \leq r'(t(\mathfrak{n})) \rightarrow$
$q_{next} := \delta(\delta(q, t(\mathfrak{n})), i)$;
$Traverse(q_{next}, \mathfrak{n} \cdot i)$
rof;
for $sp : sp \in output(\delta(q, t(\mathfrak{n}))) \rightarrow$
"register sp match at its endpoint $\mathfrak{n}$";
rof
for $sp : sp \in output(\delta(q, \nu)) \rightarrow$
"register sp match at its endpoint $\mathfrak{n}$";
rof
]];
{ Post: every stringpath match in t whose endpoint is in the subtree $t@\mathfrak{n}$
has been registered at its endpoint }

{ Pre: $M_{\text{AC}} = (Q, V' \cup \mathbb{N}_+, \delta, q_0, output)$ is the AC automaton
built on the stringpaths of the pattern tree }
$Traverse(q_0, \varepsilon)$
{ Post: every stringpath match in t has been registered at its endpoint }

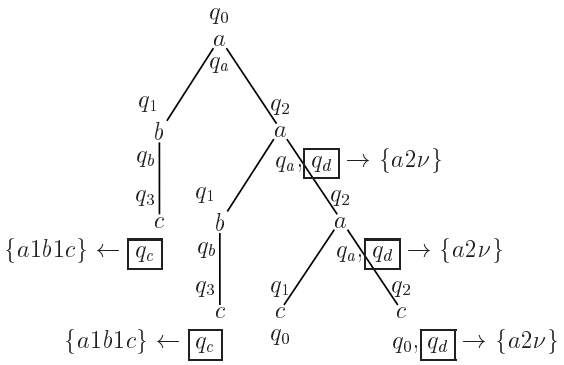


Figure 4: AC automaton state assignment and stringpath matches

Example 19. As an example, Figure 4 shows the states associated with every node and matches detected by the algorithm for subject tree $t = a(b(c), a(b(c), a(c, c)))$.

Note that even though the AC automaton used is deterministic, two states may be associated with a tree node $\mathfrak{n}$: the states corresponding to $\delta(q, t(\mathfrak{n}))$ and to $\delta(q, \nu)$. States corresponding to stringpath matches are framed. □

The algorithm can be extended to deal with multiple patterns as well, and can be used as the basis for tree acceptance and tree parsing algorithms [vdM88, AGT89].

5 Using stringpath DRFTAs

In this section, we present our new TPM algorithm. It uses a particular DRFTA and associated output function, combined with a root-to-frontier subject tree traversal.

On a high level, the DRFTA and output function construction works as follows:

1. Construct a DRFTA recognizing the pattern tree
2. For every state and alphabet symbol indicating a stringpath match, define the output of the state and symbol equal to this stringpath; for other combinations of state and alphabet symbol, define the output to be empty
3. Add 'self-loop' transitions on every symbol of rank > 0 to the start state
4. Determinize the resulting automaton and adapt the output function accordingly

The construction bears a lot of resemblance to the AC automaton construction process enumerated in the preceding section. A detailed investigation of the correspondence between the two constructions will be the subject of future work.

We discuss the above construction in more detail and show that the results can be used for root-to-frontier TPM, before presenting the new algorithm. Steps 1–3 result in a TPM NRFTA and are discussed first. In Section 5.2, step 4 is applied to obtain a DRFTA. Although this automaton cannot be used as a TPM automaton by itself, we show that it can be used for stringpath matching.

5.1 TPM NRFTA construction

Given a pattern, we can construct a DRFTA M accepting this pattern, in which the set of states is the set of dotted trees:

Construction 20. Let $p \in \mathit{Tree}(V', r')$, then $M = (Q, V', r', R, Q_{ra}, Q_{la})$ where

$$
\begin{array}{lcl}
Q & = & DT(p) \\
Q_{ra} & = & \{(p, \varepsilon)\} \\
R_a & = & \left\langle \mathbf{Set}\, \mathfrak{n} : \begin{array}{ll} (p,\mathfrak{n}) \in Q & : \begin{array}{l} ((p,\mathfrak{n}), \\ ((p,\mathfrak{n}\cdot 1), \\ \ldots, \\ (p,\mathfrak{n}\cdot r(a)))) \end{array} \\ \wedge\, p(\mathfrak{n}) = a & \end{array} \right\rangle \text{ for all } a \in V'
\end{array}
$$

□

This construction results in a *deterministic* root-to-frontier tree automaton, but when extending it to deal with multiple patterns this may no longer be the case. Note that for $\mathfrak{n}$ such that $p(\mathfrak{n})$ has rank 0, elements of R_a have the form $((p, \mathfrak{n}), ())$ i.e. relate a state (a dotted tree) to the empty tuple of states.

Example 21 (DRFTA accepting pattern p). Applying the construction to pattern $p = a(b(c), \nu)$ leads to the DRFTA depicted in Figure 5.

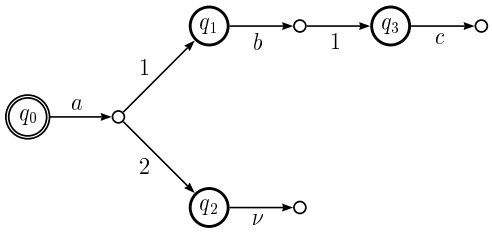


Figure 5: DRFTA resulting from Construction 20

The correspondence between the state labels used and the dotted trees they represent is as follows:

$$\begin{array}{lcl|lcl} q_0 & = & (p, \varepsilon) & q_2 & = & (p, 2) \\ q_1 & = & (p, 1) & q_3 & = & (p, 1 \cdot 1) \end{array}$$

The state assignment for every node of p in an accepting computation is shown in Figure 6. Tree p is accepted since $R_c(q_3) = ()$ and $R_\nu(q_2) = ()$. □

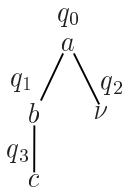


Figure 6: State assignment leading to acceptance of pattern tree p

Note how the DRFTA constructed for a tree pattern is similar to a trie constructed for the corresponding set of stringpaths.

Theorem 22. Given a subject tree t, pattern tree p, nodes $\mathfrak{m} \in D_t$ and $\mathfrak{n} \in D_p$, and a DRFTA as in Construction 20,

$$\begin{array}{l} (p, \mathfrak{n}) \text{ is assigned to node } \mathfrak{m} \text{ by} \\ \text{the DRFTA computation} \\ \wedge\ (t(\mathfrak{m}) = p(\mathfrak{n}) \vee \nu = p(\mathfrak{n})) \end{array} \Rightarrow \begin{array}{l} RPath(p, \mathfrak{n}) \text{ matches} \\ \text{ending at node } \mathfrak{m} \end{array}$$

Proof : We prove this theorem by structural induction on $\mathfrak{n}$.

Case $\mathfrak{n} = \varepsilon$: $t(\mathfrak{m}) = p(\varepsilon) \vee \nu = p(\varepsilon)$ implies that $p(\varepsilon) = RPath(p, \varepsilon)$ matches ending at node $\mathfrak{m}$.

Case $\mathfrak{n} = \mathfrak{l} \cdot i$: Using the definition of the DRFTA's transition relation, $(p, \mathfrak{n})$ is assigned to node $\mathfrak{m}$ by the DRFTA computation $\wedge\ (t(\mathfrak{m}) = p(\mathfrak{n}) \vee \nu = p(\mathfrak{n}))$ implies that $(p, \mathfrak{l})$ is assigned to node $\mathfrak{m}\downarrow 1$ and $t(\mathfrak{m}\downarrow 1) = p(\mathfrak{l})$. Using the induction hypothesis, $RPath(p, \mathfrak{l})$ matches ending at node $\mathfrak{m}\downarrow 1$. Since $t(\mathfrak{m}) = p(\mathfrak{n}) \vee \nu = p(\mathfrak{n})$, $RPath(p, \mathfrak{n}) = RPath(p, \mathfrak{l}) \cdot i \cdot p(\mathfrak{n})$ matches ending at node $\mathfrak{m}$. □

Since stringpaths are rootpaths ending in symbols of rank 0, matches can only end in such symbols, and using Theorem 22 we obtain the following definition:

Definition 23. For automata as in Construction 20, partial function $output \in Q \times V' \to (V' \cdot \mathbb{N}_+)^* \cdot V'$ is defined for $(p, \mathfrak{n}) \in Q$ and $a \in V'$ such that $a = p(\mathfrak{n}) \wedge r(a) = 0$ by $output((p, \mathfrak{n}), a) = RPath(p, \mathfrak{n})$. □

Note that function *output* is used with the symbol $t(\mathfrak{m})$ for $\mathfrak{m}$ the node of t that a state is assigned to, and with symbol ν. The inverse of the implication in Theorem 22 does not hold; the automaton is a tree *acceptor*, and can only be used to detect a pattern match that starts at the subject tree root. To enable pattern matches starting at other input tree nodes to be detected, an extension similar to the addition of the 'self-loop' transitions of the AC automaton is necessary, as follows:

Construction 24. Let $p \in Tree(V', r')$, then $M' = (Q, V', r', R', Q_{ra}, Q_{la})$ where

$$R'_a = R_a \cup \begin{cases} \{((p, \varepsilon), ((p, \varepsilon)^{r(a)}))\} & \text{for all } a \in V' \text{ with } r(a) > 0 \\ \varnothing & \text{for all } a \in V' \text{ with } r(a) = 0 \end{cases}$$

□

The result is an NRFTA accepting all trees ending in pattern occurrences.

Example 25 (Stringpath NRFTA for pattern p). The NRFTA with 'self-loops' constructed for pattern $p = a(b(c), \nu)$ by Construction 24—corresponding to steps 1–3 of the high-level construction at the beginning of Section 5—is depicted in Figure 7. The output function is defined by $output(q_3, c) = a1b1c$, $output(q_2, \nu) = a2\nu$ and undefined for other input range values. □

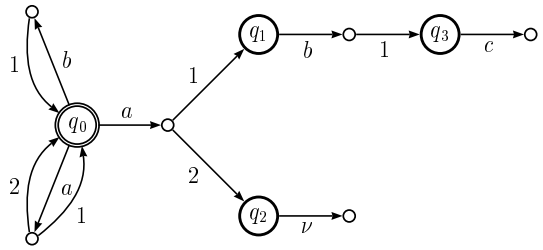


Figure 7: NRFTA with 'self-loops' resulting from Construction 24

Theorem 26. Given a subject tree t, pattern tree p, nodes $\mathfrak{m} \in D_t$ and $\mathfrak{n} \in D_p$, and an NRFTA as in Construction 24,

$$\begin{array}{l} (p, \mathfrak{n}) \text{ is assigned to node } \mathfrak{m} \text{ by an} \\ \text{NRFTA computation} \\ \wedge\ (t(\mathfrak{m}) = p(\mathfrak{n}) \vee \nu = p(\mathfrak{n})) \end{array} \equiv \begin{array}{l} RPath(p, \mathfrak{n}) \text{ matches} \\ \text{ending at node } \mathfrak{m} \end{array}$$

Proof : $\Rightarrow$: As in the proof of Theorem 22. $\Leftarrow$: By structural induction on $\mathfrak{n}$.

Case $\mathfrak{n} = \varepsilon$: $RPath(p, \varepsilon) = p(\varepsilon)$ matches ending at node $\mathfrak{m}$ implies that $t(\mathfrak{m}) = p(\varepsilon) \vee \nu = p(\varepsilon)$. From the definition of the NRFTA's transition function, (p, ε) is assigned to any given node in some computation of the NRFTA.

Case $\mathfrak{n} = \mathfrak{l} \cdot i$: $RPath(p, \mathfrak{n})$ matches ending at node $\mathfrak{m}$ implies that $RPath(p, \mathfrak{l})$ matches ending at node $\mathfrak{m}\lfloor 1$ and $t(\mathfrak{m}) = p(\mathfrak{n}) \vee \nu = p(\mathfrak{n})$. Using the induction hypothesis, $(p, \mathfrak{l})$ is assigned to node $\mathfrak{m}\lfloor 1$ by a computation of the NRFTA $\wedge\ (t(\mathfrak{m}\lfloor 1) = p(\mathfrak{l}) \vee \nu = p(\mathfrak{l}))$. Since $r(\nu) = 0$, the second conjunct reduces to $t(\mathfrak{m}\lfloor 1) = p(\mathfrak{l})$, and using the transition function definition we have that $(p, \mathfrak{l} \cdot i) = (p, \mathfrak{n})$ is assigned to node $\mathfrak{m}$ by a computation of the NRFTA. Since we already had $t(\mathfrak{m}) = p(\mathfrak{n}) \vee \nu = p(\mathfrak{n})$ this completes the proof of this case. □

5.2 Determinization

Similarly to the determinization of the trie with 'self-loops' to obtain a deterministic AC automaton, the NRFTA resulting from steps 1–3 can be determinized. The subset construction for NRFTAs is a straightforward generalization of that for string automata and is not elaborated here. It is known from regular tree theory however that the resulting DRFTA in general recognizes a superset of the NRFTA's language: the set of trees of which every stringpath occurs as a stringpath in a tree from the NRFTA's language. We aim at using the resulting DRFTA for tree stringpath pattern matching however, and it turns out to be suitable for this purpose.

Example 27 (Stringpath DRFTA for pattern p). Applying the subset construction to the NRFTA of Example 25 (corresponding to step 4 at the beginning of Section 5) leads to the stringpath DRFTA depicted in Figure 8. Output function values for this example DRFTA are singleton set versions of the values for the NRFTA of Example 25. As in Example 18, states of the automaton are different from those with the same label in the automaton of Example 25. □

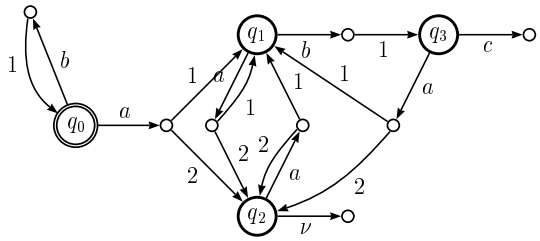


Figure 8: Stringpath DRFTA for p. Missing transitions on symbols of rank > 0 lead to (tuples of size equal to the symbol's rank of) state q_0

Using Theorem 26 and the subset construction, we obtain:

Corollary 28. Given a subject tree t, pattern tree p, nodes $\mathfrak{m} \in D_t$ and $\mathfrak{n} \in D_p$, and a DRFTA obtained from Construction 24 by a subset construction,

$$\begin{array}{l}(p,\mathfrak{n}) \text{ is part of the state assigned} \\ \text{to node } \mathfrak{m} \text{ by the DRFTA computation} \\ \wedge\ ((t(\mathfrak{m}) = p(\mathfrak{n})) \vee (\nu = p(\mathfrak{n})))\end{array} \equiv \begin{array}{l}RPath(p,\mathfrak{n}) \text{ matches} \\ \text{ending at node } \mathfrak{m}\end{array}$$

□

In other words, the state assigned to a node and the symbol at that node and ν together determine the set of all matching stringpaths ending at that node. As indicated before, the DRFTA and associated output function can thus be used in a root-to-frontier subject tree traversal to detect all stringpath matches.

5.3 A DRFTA-based TPM algorithm

As an invariant, when visiting a node $\mathfrak{n}$ of the given subject tree t, the DRFTA is in the state assigned to the node based on the symbols on the rootpath $RPath(t,\mathfrak{n})$ with the exception of the last symbol of this rootpath—symbol $t(\mathfrak{n})$.

As in the previous algorithm, the recursive procedure is called on every child i of the current node, with a state obtained by projecting away all except the ith component of the state tuple reached by a transition from the current state on $t(\mathfrak{n})$.

When visiting a node, the algorithm should register any matches indicated by the DRFTA's *output* function for the current state and either symbol $t(\mathfrak{n})$ or ν, since ν matches any subtree. This leads to the following algorithm:

{ Pre: $q = p_n$ where $n = |\mathfrak{n}| \wedge p_0 = q_0 \wedge$
$\langle \forall\, i :\ 1 \le i \le n\ :\ p_i = \pi_{RPath(t,\mathfrak{n})_{2i}}(R_{RPath(t,\mathfrak{n})_{2i-1}}(p_{i-1})) \rangle$ }
proc $Traverse(q : Q, \mathfrak{n} : D) =$
[[**var** $q_{next} : Q;\ i : \mathbb{N}_+ ; sp : (V \cdot \mathbb{N}_+)^* \cdot V$
| **for** $i : 1 \le i \le r'(t(\mathfrak{n})) \rightarrow$
$q_{next} := \pi_i(R_{t(\mathfrak{n})}(q));$
$Traverse(q_{next}, \mathfrak{n} \cdot i)$
rof;
for $sp : sp \in output(q, t(\mathfrak{n})) \rightarrow$
"register sp match at its endpoint $\mathfrak{n}$";
rof
for $sp : sp \in output(q, \nu) \rightarrow$
"register sp match at its endpoint $\mathfrak{n}$";
rof
]];
{ Post: every stringpath match in t whose endpoint is in the subtree $t@\mathfrak{n}$
has been registered at its endpoint }

{ Pre: $M_{\text{DRFTA}} = (Q, V', r', R, q_0, Q_{la})$ is the DRFTA built on the
pattern tree, and *output* is the associated output function }
$Traverse(q_0, \varepsilon)$
{ Post: every stringpath match in t has been registered at its endpoint }

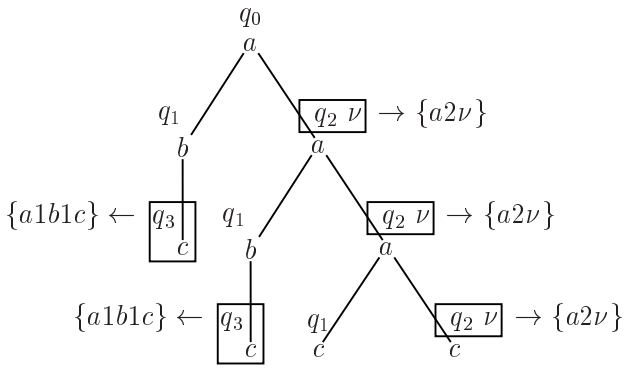


Figure 9: DRFTA state assignment and stringpath matches

Example 29. Figure 9 shows the states associated with every node and matches detected by the algorithm for $t = a(b(c), a(b(c), a(c, c)))$. Combinations of states and symbols corresponding to stringpath matches are framed. Note that symbol ν is only explicitly depicted for nodes at which it occurs in a stringpath match. □

6 Concluding remarks

We presented two algorithms for stringpath-based tree pattern matching. One of these, based on a root-to-frontier tree traversal and using an Aho-Corasick automaton, is already well known from the literature [HO82, AGT89, vdM88, AG85]. The other, based on a root-to-frontier tree traversal and using a DRFTA, is new. By presenting the two in a similar style, we highlighted their similarities and provided a missing link between TPM algorithms using tree automata and those using stringpath automata.

The two TPM algorithms are very similar, their difference being restricted to the different automata and output functions used. As future work, we intend to compare the automata in more detail. We conjecture that they are in some sense equivalent, i.e. can be transformed into one another.

We intend to extend the new algorithm to multiple tree patterns and from there to a tree acceptance and a tree parsing algorithm, providing related solutions to the related problems of tree acceptance and tree parsing. The result will likely be similar to the Aho-Corasick-based tree acceptance and tree parsing algorithms of Aho, Ganapathi & Tjiang [AGT89, vdM88, AG85].

Finally, it would be interesting to investigate the use of different keyword pattern matching automata or algorithms—such as those in [CWZ04, Wat95]—to obtain new tree pattern matching algorithms that are based on stringpath matching. One such algorithm, using Boyer-Moore pattern matching, was presented in [Wat97].

References

[AC75] A.V. Aho and M.J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18:333–340, 1975.

[AG85] A.V. Aho and M. Ganapathi. Efficient tree pattern matching: An aid to code generation. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages*, pages 334–340, 1985.

[AGT89] A.V. Aho, M. Ganapathi, and S.W.K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, 1989.

[CH97] R. Cole and R. Hariharan. Tree pattern matching and subset matching in randomized o(n log3 m) time. In *Proceedings of the ACM Symposium on Theory of Computing*, pages 66–75, 1997.

[Cha87] David R. Chase. An improvement to bottom-up tree pattern matching. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 168–177. ACM, 1987.

[CHI99] R. Cole, R. Hariharan, and P. Indyk. Tree pattern matching and subset matching in deterministic o($nlog^3n$) time. In *Proceedings of the 10th ACM-SIAM Symposium on Discrete Algorithms*, pages 245–254, 1999.

[CR03] Maxime Crochemore and Wojciech Rytter. *Jewels of Stringology - Text Algorithms*. World Scientific Publishing, 2003.

[CWZ04] Loek Cleophas, Bruce W. Watson, and Gerard Zwaan. Automaton-based sublinear keyword pattern matching. In *Proceedings of the 11th international conference on String Processing and Information REtrieval (SPIRE 2004)*, volume 3246 of *LNCS*. Springer, October 2004.

[DGM94] M. Dubiner, Z. Galil, and E. Magen. Faster tree pattern matching. *Journal of the ACM*, 41(2):205–213, 1994.

[Eng75] Joost Engelfriet. Tree Automata and Tree Grammars. Lecture Notes DAIMI FN-10, Aarhus University, April 1975.

[FSW94] Christian Ferdinand, Helmut Seidl, and Reinhard Wilhelm. Tree automata for code selection. *Acta Informatica*, 31:741–760, 1994.

[GS97] Ferenc Gécseg and Magnus Steinby. *Tree Languages*, volume 3 of *Handbook of Formal Languages*, pages 1–68. Springer, 1997.

[HC86] Philip J. Hatcher and Thomas W. Christopher. High-quality code generation via bottom-up tree pattern matching. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 119–130. ACM, 1986.

[HK89] C. Hemerik and J.P. Katoen. Bottom-up tree acceptors. *Science of Computer Programming*, 13(1):51–72, 1989.

[HO82] C.M. Hoffmann and M.J. O'Donnell. Pattern matching in trees. *Journal of the ACM*, 29(1):68–95, January 1982.

[Kos89] S.R. Kosaraju. Efficient tree pattern matching. In *Proceedings of the 30th annual IEEE Symposium on Foundations of Computer Science, FOCS'89*, pages 178–183. IEEE Computer Society Press, 1989.

[Kro75] H. Kron. *Tree templates and subtree transformational grammars*. PhD thesis, University of California, Santa Cruz, 1975.

[NR02] Gonzalo Navarro and Mathieu Raffinot. *Flexible pattern matching in strings: practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, 2002.

[vD87] Yolanda van Dinther. De systematische afleiding van acceptoren en ontleders voor boom-grammatica's. Master's thesis, Faculteit Wiskunde en Informatica, Technische Universiteit Eindhoven, August 1987. (In Dutch).

[vdM88] H.J.A. van de Meerakker. Een parsing algoritme voor boomgrammatica's. Master's thesis, Faculteit Wiskunde en Informatica, Technische Universiteit Eindhoven, May 1988. (In Dutch).

[Wat95] Bruce W. Watson. *Taxonomies and Toolkits of Regular Language Algorithms*. PhD thesis, Technische Universiteit Eindhoven, September 1995.

[Wat97] Bruce W. Watson. A Boyer-Moore (or Watson-Watson) Type Algorithm for Regular Tree Pattern Matching. In *Proceedings of the Prague Stringology Club Workshop '97*, pages 33–38, 1997.

A Simple Alphabet-Independent FM-Index

Szymon Grabowski[1], Veli Mäkinen[2*],
Gonzalo Navarro[3†], and Alejandro Salinger[3]

[1] Computer Engineering Dept., Tech. Univ. of Łódź, Poland.
e-mail: sgrabow@zly.kis.p.lodz.pl

[2] Technische Fakultät, Bielefeld Universität, Germany
e-mail: veli@cebitec.uni-bielefeld.de

[3] Dept. of Computer Science, Univ. of Chile, Chile.
e-mail: {gnavarro,asalinge}@dcc.uchile.cl

Abstract. We design a succinct full-text index based on the idea of Huffman-compressing the text and then applying the Burrows-Wheeler transform over it. The resulting structure can be searched as an FM-index, with the benefit of removing the sharp dependence on the alphabet size, σ, present in that structure. On a text of length n with zero-order entropy H_0, our index needs $O(n(H_0+1))$ bits of space, without any dependence on σ. The average search time for a pattern of length m is $O(m(H_0+1))$, under reasonable assumptions. Each position of a text occurrence can be reported in worst case time $O((H_0+1)\log n)$, while any text substring of length L can be retrieved in $O((H_0+1)L)$ average time in addition to the previous worst case time. Our index provides a relevant space/time tradeoff between existing succinct data structures, with the additional interest of being easy to implement. Our experimental results show that, although not among the most succinct, our index is faster than the others in many aspects, even letting them use significatively more space.

1 Introduction

A *full-text index* is a data structure that enables to determine the *occ* occurrences of a short pattern $P = p_1 p_2 \ldots p_m$ in a large text $T = t_1 t_2 \ldots t_n$ without a need of scanning over the whole text T. Text and pattern are sequences of characters over an alphabet Σ of size σ. In practice one wants to know not only the value *occ*, i.e., how many times the pattern appears in the text (*counting query*) but also the text positions of those *occ* occurrences (*reporting query*), and usually also a text context around them (*display query*).

A classic example of a full-text index is the *suffix tree* [20] reaching $O(m + occ)$ time complexity for counting and reporting queries. Unfortunately, it takes $O(n \log n)$ bits,[1] and also the constant factor is large. A smaller space complexity factor is achieved by the *suffix array* [13], reaching $O(m \log n + occ)$ or $O(m + \log n + occ)$ in

*Funded by the Deutsche Forschungsgemeinschaft (BO 1910/1-3) within the Computer Science Action Program.

†Funded in part by Fondecyt Grant 1-050493 (Chile).

[1] By log we mean $\log_2$ in this paper.

time (depending on a variant), but still the space usage may rule out this structure from some applications, e.g. in computational biology.

The large space requirement of traditional full-text indexes has raised a natural interest in *succinct* full-text indexes that achieve good tradeoffs between search time and space complexity [12, 3, 10, 19, 8, 15, 18, 16, 9]. A truly exciting perspective has been originated in the work of Ferragina and Manzini [3]; they showed a full-text index may discard the original text, as it contains enough information to recover the text. We denote a structure with such a property with the term *self-index*.

The FM-index of Ferragina and Manzini [3] was the first self-index with space complexity expressed in terms of kth order (empirical) entropy and pattern search time linear only in the pattern length. Its space complexity, however, contains an exponential dependence on the alphabet size; a weakness eliminated in a practical implementation [4] for the price of not achieving the optimal search time anymore. Therefore, it has been interesting both from the point of theory and practice to construct an index with nice bounds both in space and time complexities, preferably with no (or mild) dependence on the alphabet size.

In this paper we concentrate on improving the FM-index, in particular its large alphabet dependence. This dependence shows up not only in the space usage, but also in the time to show an occurrence position and display text substrings. The FM-index needs up to $5H_k n + O\left((\sigma \log \sigma + \log\log n)\frac{n}{\log n} + n^\gamma \sigma^{\sigma+1}\right)$ bits of space, where $0 < \gamma < 1$. The time to search for a pattern and obtain the number of its occurrences in the text is the optimal $O(m)$. The text position of each occurrence can be found in $O\left(\sigma \ \log^{1+\varepsilon} n\right)$ time, for some $\varepsilon > 0$ that appears in the sublinear terms of the space complexity. Finally, the time to display a text substring of length L is $O\left(\sigma \ (L + \log^{1+\varepsilon} n)\right)$. The last operation is important not only to show a text context around each occurrence, but also because a self-index replaces the text and hence it must provide the functionality of retrieving any desired text substring.

The compressed suffix array (CSA) of Sadakane [19] can be seen as a tradeoff with larger search time but much milder dependence on the alphabet size. The CSA needs $(H_0/\varepsilon + O(\log\log\sigma))n$ bits of space. Its search time (finding the number of occurrences of a pattern) is $O(m \log n)$. Each occurrence can be reported in $O\left(\log^\varepsilon n\right)$ time, and a text substring of length L can be displayed in $O\left(L + \log^\varepsilon n\right)$ time.

In this paper we present a simple structure based on the FM-index concept. We Huffman-compress the text and then apply the Burrows-Wheeler transform over it, as in the FM-index. The obtained structure can be regarded as an FM-index built over a binary sequence. As a result, we remove any dependence on the alphabet size. We show that our index can operate using $n(2H_0+3+\varepsilon)(1+o(1))$ bits, for any $\varepsilon > 0$. No alphabet dependence is hidden in the sublinear terms.

At search time, our index finds the number of occurrences of the pattern in $O(m(H_0+1))$ average time. The text position of each occurrence can be reported in worst case time $O\left(\frac{1}{\varepsilon}(H_0+1) \ \log n\right)$. Any text substring of length L can be displayed in $O\left((H_0+1) \ L\right)$ average time, in addition to the mentioned worst case time to find a text position. In the worst case all the H_0 become $\log n$.

This index was first presented in a poster [5], where we only gave its rough idea. Now we present it in full detail and explore its empirical effectiveness in counting, reporting and displaying, for a broad scope of real-world data (English text, DNA and proteins). We also include a k-ary Huffman variant. We show that our index,

Algorithm FM_Search(P,T^{bwt})
(1) $i = m$;
(2) $sp = 1$; $ep = n$;
(3) **while** (($sp \leq ep$) **and** ($i \geq 1$)) **do**
(4) $\quad c = P[i]$;
(5) $\quad sp = C[c] + Occ(T^{bwt}, c, sp-1)+1$;
(6) $\quad ep = C[c] + Occ(T^{bwt}, c, ep)$;
(7) $\quad i = i - 1$;
(8) **if** ($ep < sp$) **then return** "not found" **else return** "found ($ep - sp + 1$) occs".

Figure 1: Algorithm for counting the number of occurrences of $P[1 \ldots m]$ in $T[1 \ldots n]$.

albeit not among the most succinct indexes, is faster than the others in many cases, even if we give the other indexes much more space to work.

2 The FM-index Structure

The FM-index [3] is based on the *Burrows-Wheeler transform (BWT)* [1], which produces a permutation of the original text, denoted by $T^{bwt} = bwt(T)$. String T^{bwt} is the result of the following *forward* transformation: (1) Append to the end of T a special end marker \$, which is lexicographically smaller than any other character; (2) form a *conceptual* matrix $\mathcal{M}$ whose rows are the cyclic shifts of the string $T\$$, sorted in lexicographic order; (3) construct the transformed text L by taking the last column of $\mathcal{M}$. The first column is denoted by F.

The *suffix array (SA)* $\mathcal{A}$ of text $T\$$ is essentially the matrix $\mathcal{M}$: $\mathcal{A}[i] = j$ iff the ith row of $\mathcal{M}$ contains string $t_j t_{j+1} \cdots t_n \$ t_1 \cdots t_{j-1}$. The occurrences of any pattern $P = p_1 p_2 \cdots p_m$ form an interval $[sp, ep]$ in $\mathcal{A}$, such that suffixes $t_{\mathcal{A}[i]} t_{\mathcal{A}[i]+1} \cdots t_n$, $sp \leq i \leq ep$, contain the pattern as a prefix. This interval can be searched for by using two binary searches in time $O(m \log n)$.

The suffix array of text T is represented implicitly by T^{bwt}. The novel idea of the FM-index is to store T^{bwt} in compressed form, and to simulate the search in the suffix array. To describe the search algorithm, we need to introduce the *backward* BWT that produces T given T^{bwt}: (*i*) Compute the array $C[1 \ldots \sigma]$ storing in $C[c]$ the number of occurrences of characters $\{\$, 1, \ldots, c-1\}$ in the text T. Notice that $C[c] + 1$ is the position of the first occurrence of c in F (if any). (*ii*) Define the *LF-mapping* $LF[1 \ldots n+1]$ as $LF[i] = C[L[i]] + Occ(L, L[i], i)$, where $Occ(X, c, i)$ equals the number of occurrences of character c in the prefix $X[1, i]$. (*iii*) Reconstruct T backwards as follows: set $s = 1$ and $T[n] = L[1]$ (because $\mathcal{M}[1] = \$T$); then, for each $n-1, \ldots, 1$ do $s \leftarrow LF[s]$ and $T[i] \leftarrow L[s]$.

We are now ready to describe the search algorithm given in [3] (Fig. 1). It finds the interval of $\mathcal{A}$ containing the occurrences of the pattern P. It uses the array C and function $Occ(X, c, i)$ defined above. Using the properties of the backward BWT, it is easy to see that the algorithm maintains the following invariant [3]: *At the ith phase, with i from m to 1, the variable sp points to the first row of $\mathcal{M}$ prefixed by $P[i, m]$ and the variable ep points to the last row of $\mathcal{M}$ prefixed by $P[i, m]$.* The correctness of the algorithm follows from this observation.

Ferragina and Manzini [3] describe an implementation of $Occ(T^{bwt}, c, i)$ that uses a compressed form of T^{bwt}. They show how to compute $Occ(T^{bwt}, c, i)$ for any c and i in constant time. However, to achieve this they need exponential space (in the size of the alphabet). In a practical implementation [4] this was avoided, but the constant time guarantee for answering $Occ(T^{bwt}, c, i)$ was no longer valid.

The FM-index can also show the text positions where P occurs, and display any text substring. The details are deferred to Section 4.

3 First Huffman, then Burrows-Wheeler

We focus now on our index representation. Imagine that we compress our text $T\$$ using Huffman. The resulting bit stream will be of length $n' < (H_0 + 1)n$, since (binary) Huffman poses a maximum representation overhead of 1 bit per symbol[2]. We call T' this sequence, and define a second bit array Th, of the same length of T', such that $Th[i] = 1$ iff i is the starting position of a Huffman codeword in T'. Th is also of length n'. (We will not represent T' nor Th in our index.)

The idea is to search the binary text T' instead of the original text T. Let us apply the Burrows-Wheeler transform over text T', so as to obtain $B = (T')^{bwt}$. In order to have a binary alphabet, T' will not have its own special terminator character "\$" (yet that of T is encoded in binary at the end of T').

More precisely, let $\mathcal{A}'[1 \ldots n']$ be the suffix array for text T', that is, a permutation of the set $1 \ldots n'$ such that $T'[A'[i] \ldots n'] < T'[A'[i+1] \ldots n']$ in lexicographic order, for all $1 \leq i < n'$. In a lexicographic comparison, if a string x is a prefix of y, assume $x < y$. Suffix array $\mathcal{A}'$ will not be explicitly represented. Rather, we represent bit array $B[1 \ldots n']$, such that $B[i] = T'[\mathcal{A}'[i] - 1]$ (except that $B[i] = T[n']$ if $\mathcal{A}'[i] = 1$). We also represent another bit array $Bh[1 \ldots n']$, such that $Bh[i] = Th[\mathcal{A}'[i]]$. This tells whether position i in $\mathcal{A}'$ points to the beginning of a codeword.

Our goal is to search B exactly like the FM-index. For this sake we need array C and function Occ. Since the alphabet is binary, however, Occ can be easily computed: $Occ(B, 1, i) = rank(B, i)$ and $Occ(B, 0, i) = i - rank(B, i)$, where $rank(B, i)$ is the number of 1's in $B[1 \ldots i]$, $rank(B, 0) = 0$. This function can be computed in constant time using only $o(n)$ extra bits [11, 14, 2]. The solution, as well as its more practical implementation variants, are described in [7].

Also, array C is so simple for the binary text that we can do without it: $C[0] = 0$ and $C[1] = n' - rank(B, n')$, that is, the number of zeros in B (of course value $n' - rank(B, n')$ is precomputed). Therefore, $C[c] + Occ(T^{bwt}, c, i)$ is replaced in our index by $i - rank(B, i)$ if $c = 0$ and $n' - rank(B, n') + rank(B, i)$ if $c = 1$.

There is a small twist, however, due to the fact that we are not putting a terminator to our binary sequence T' and hence no terminator appears in B. Let us call "#" the terminator of the binary sequence so that it is not confused with the terminator "\$" of $T\$$. In the position $p_\#$ such that $\mathcal{A}'[p_\#] = 1$, we should have $B[p_\#] = \#$. Instead, we are setting $B[p_\#]$ to the last bit of T'. This is the last bit of the Huffman codeword assigned to the terminator "\$" of $T\$$. Since we can freely switch left and right siblings in the Huffman code, we will ensure that this last bit is zero. Hence the

[2] Note that these n and H_0 refer to $T\$$, not T. However, the difference between both is only $O(\log n)$, and will be absorbed by the $o(n)$ terms that will appear later.

Algorithm Huff-FM_Search(P',B,Bh)
(1) $i = m'$;
(2) $sp = 1$; $ep = n'$;
(3) **while** (($sp \leq ep$) **and** ($i \geq 1$)) **do**
(4) **if** $P'[i] = 0$ then
$sp = (sp - 1) - rank(B, sp - 1) + 1 + [sp - 1 < p_{\#}]$;
$ep = ep - rank(B, ep) + [ep < p_{\#}]$;
else $sp = n' - rank(B, n') + rank(B, sp - 1) + 1$;
$ep = n' - rank(B, n') + rank(B, ep)$;
(7) $i = i - 1$;
(8) **if** $ep < sp$ **then** $occ = 0$ **else** $occ = rank(Bh, ep) - rank(Bh, sp - 1)$;
(9) **if** $occ = 0$ **then return** "not found" **else return** "found (occ) occs".

Figure 2: Algorithm for counting the number of occurrences of $P'[1 \ldots m']$ in $T'[1 \ldots n']$.

correct B sequence would be of length $n' + 1$, starting with 0 (which corresponds to $T'[n']$, the character preceding the occurrence of "#", since $\# < 0 < 1$), and it would have $B[p_{\#}] = \#$. To obtain the right mapping to our binary B, we must correct $C[0] + Occ(B, 0, i) = i - rank(B, i) + [i < p_{\#}]$, that is, add 1 to the original value when $i < p_{\#}$. The computation of $C[1] + Occ(B, 1, i)$ remains unchanged.

Therefore, by preprocessing B to solve $rank$ queries, we can search B exactly as the FM-index. The extra space required by the $rank$ structure is $o(H_0 n)$, without any dependence on the alphabet size. Overall, we have used at most $n(2H_0 + 2)(1 + o(1))$ bits for our representation. This will grow slightly in the next sections due to additional requirements.

Our search pattern is not the original P, but its binary coding P' using the Huffman code we applied to T. If we assume that the characters in P have the same distribution of T, then the length of P' is $< m(H_0 + 1)$. This is the number of steps to search B using the FM-index search algorithm.

The answer to that search, however, is different from that of the search of T for P. The reason is that the search of T' for P' returns the number of suffixes of T' that start with P'. Certainly these include the suffixes of T that start with P, but also other superfluous occurrences may appear. These correspond to suffixes of T' that do not start a Huffman codeword, yet they start with P'.

This is why we have marked the suffixes that start a Huffman codeword in Bh. In the range $[sp, ep]$ found by the search for P' in B, every bit set in $Bh[sp \ldots ep]$ represents a true occurrence. Hence the true number of occurrences can be computed as $rank(Bh, ep) - rank(Bh, sp - 1)$. Figure 2 shows the search algorithm.

Therefore, the search complexity is $O(m(H_0 + 1))$, assuming that the zero-order distributions of P and T are similar. Next we show that the worst case search cost is $O(m \log n)$. This matches the worst case search cost of the original CSA (while our average case is better).

For the worst case, we must determine which is the maximum height of a Huffman tree with total frequency n. Consider the longest root-to-leaf path in the Huffman tree. The leaf symbol has frequency at least 1. Let us traverse the path upwards and consider the (sum of) frequencies encountered in the other branch at each node.

These numbers must be, at least: 1, 1, 2, 3, 5, ..., that is, the Fibonacci sequence $F(i)$. Hence, a Huffman tree with depth d needs that the text is of length at least $n \geq 1 + \sum_{i=1}^{d} F(i) = F(d+2)$ [21, pp. 397]. Therefore, the maximum length of a code is $F^{-1}(n) - 2 = \log_\phi(n) - 2 + o(1)$, where $\phi = (1+\sqrt{5})/2$.

Therefore, the encoded pattern P' cannot be longer than $O(m \log n)$ and this is also the worst case search cost, as promised. An exception to the above argument occurs when P contains a character not present in T. This is easier, however, as we immediately know that P does not occur in T.

Actually, it is possible to reduce the worst-case search time to $O(m \log \sigma)$, without altering the average search time nor the space usage, by forcing the Huffman tree to become balanced after level $(1+x)\log\sigma$. For details see [6].

4 Reporting Occurrences and Displaying the Text

Up to now we have focused on the search time, that is, the time to determine the suffix array interval containing all the occurrences. In practice, one needs also the text positions where they appear, as well as a text context. Since self-indexes replace the text, in general one needs to extract any text substring from the index.

Given the suffix array interval that contains the *occ* occurrences found, the FM-index reports each such position in $O(\sigma \log^{1+\varepsilon} n)$ time, for any $\varepsilon > 0$ (which appears in the sublinear space component). The CSA can report each in $O(\log^{\varepsilon} n)$ time, where ε is paid in the nH_0/ε space. Similarly, a text substring of length L can be displayed in time $O(\sigma(L + \log^{1+\varepsilon} n))$ by the FM-index and $O(L + \log^{\varepsilon} n)$ by the CSA.

In this section we show that our index can do better than the FM-index, although not as well as the CSA. Using $(1+\varepsilon)n$ additional bits, we can report each occurrence position in $O(\frac{1}{\varepsilon}(H_0+1)\log n)$ time and display a text context in time $O(L \log\sigma + \log n)$ in addition to the time to find an occurrence position. On average, assuming that random text positions are involved, the overall complexity to display a text interval becomes $O((H_0+1)(L + \frac{1}{\varepsilon}\log n))$.

4.1 Reporting Occurrences

A first problem is how to extract, in $O(occ)$ time, the *occ* positions of the bits set in $Bh[sp \ldots ep]$. This is easy using *select* function: $select(Bh, j)$, gives the position of the j-th bit set in Bh. This is the inverse of function *rank* and it can also be implemented in constant time using $o(n)$ additional space [11, 14, 2, 7]. Actually we need a simpler version, $selectnext(Bh, j)$, which gives the first 1 in $Bh[j, n]$.

Let $r = rank(Bh, sp-1)$. Then, the positions of the bits set in Bh are $select(Bh, r+1)$, $select(Bh, r+2)$, ..., $select(Bh, r+occ)$. We recall that $occ = rank(Bh, ep) - rank(Bh, sp-1)$. This can be expressed using *selectnext*: The positions $pos_1 \ldots pos_{occ}$ can be found as $pos_1 = selectnext(Bh, sp)$, and $pos_{i+1} = selectnext(Bh, pos_i + 1)$. We focus now on how to find the text position of a valid occurrence.

We choose some $\varepsilon > 0$ and sample $\lfloor \frac{\varepsilon n}{2 \log n} \rfloor$ positions of T' at regular intervals, with the restriction that only codeword beginnings can be chosen. For this sake, pick positions in T' at regular intervals of length $\ell = \lceil \frac{2n'}{\varepsilon n} \log n \rceil$, and for each such position $1 + \ell(i-1)$, choose the beginning of the codeword being represented at $1 + \ell(i-1)$.

Recall from Section 3 that no Huffman codeword can be longer than $\log_\phi n - 2 + o(1)$

bits. Then, the distance between two chosen positions in T', after the adjustment, cannot exceed

$$\ell + \log_\phi n - 2 + o(1) \quad \leq \quad \frac{2}{\varepsilon}(H_0+1)\log n + \log_\phi n - 1 + o(1) \quad = \quad O\left(\frac{1}{\varepsilon}(H_0+1)\log n\right)$$

Now, store an array TS with the $\lfloor \frac{\varepsilon n}{2\log n} \rfloor$ positions of $\mathcal{A}'$ pointing to the chosen positions of T', in increasing text position order. More precisely, $TS[i]$ refers to position $1+\ell(i-1)$ in T' and hence $TS[i] = j$ such that $A'[j] = select(Th, rank(Th, 1+\ell(i-1)))$. Array TS requires $\frac{\varepsilon n}{2}(1+o(1))$ bits, since each entry needs $\log n' \leq \log(n \log\min(n,\sigma)) = \log n + O(\log\log\min(n,\sigma))$ bits.

The same $\mathcal{A}'$ positions are now sorted and the corresponding T positions (that is, $rank(Th, \mathcal{A}'[i])$) are stored in array ST, for other $\frac{\varepsilon n}{2}$ bits. Finally, we store an array S of n bits so that $S[i] = 1$ iff $\mathcal{A}'[select(Bh, i)]$ is in the sampled set. That is, $S[i]$ tells whether the i-th entry of $\mathcal{A}'$ pointing to beginning of codewords, points to a sampled text position. S is further processed for $rank$ queries.

Overall, we spend $(1+\varepsilon)n(1+o(1))$ bits for these three arrays, raising our final space requirement to $n(2H_0+3+\varepsilon)(1+o(1))$.

Let us focus first in how to determine the text position corresponding to an entry $\mathcal{A}'[i]$ for which $Bh[i] = 1$. Use bit array $S[rank(Bh, i)]$ to determine whether $\mathcal{A}'[i]$ points or not to a codeword beginning in T' that has been sampled. If it does, then find the corresponding T position in $ST[rank(S, rank(Bh, i))]$ and we are done. Otherwise, just as with the FM-index, determine position i' whose value is $\mathcal{A}'[i'] = \mathcal{A}'[i]-1$. Repeat this process, which corresponds to moving backward bit by bit in T', until a new codeword beginning is found, that is, $Bh[i'] = 1$. Now determine again whether i' corresponds to a sampled character in T: Use $S[rank(Bh, i')]$ to determine whether $\mathcal{A}'[i']$ is present in ST. If it is, report text position $1 + ST[rank(S, rank(Bh, i'))]$ and finish. Otherwise, continue with i'' trying to report $2 + ST[rank(S, rank(Bh, i''))]$, and so on. The process must finish after $O\left(\frac{1}{\varepsilon}(H_0+1)\log n\right)$ backward steps in T' because we are considering consecutive positions of T' and that is the maximum distance among consecutive samples.

We have to specify how we determine i' from i. In the FM-index, this is done via the LF-mapping, $i' = C[T^{bwt}[i]] + Occ(T^{bwt}, T^{bwt}[i], i)$. In our index, the LF-mapping over $\mathcal{A}'$ is implemented as $i' = i - rank(B, i)$ if $B[i] = 0$ and $i' = n' - rank(B, n') + rank(B, i)$ if $B[i] = 1$. This LF-mapping moves us from position $T'[\mathcal{A}'[i]]$ to $T'[\mathcal{A}'[i]-1]$.

Overall, an occurrence can be reported in worst case time $O(\frac{1}{\varepsilon}(H_0+1)\log n)$. Figure 3 gives the pseudocode.

4.2 Displaying Text

In order to display a text substring $T[l \ldots r]$ of length $L = r - l + 1$, we start by binary searching TS for the smallest sampled text position larger than r. Given value $TS[j]$, we know that $S[rank(Bh, TS[j])] = 1$ as it is a sampled $\mathcal{A}'$ entry, and the corresponding T position is simply $ST[rank(S, rank(Bh, TS[j]))]$. Once we find the first sampled text position that follows r, we have its corresponding position $i = TS[j]$ in $\mathcal{A}'$. From there on, we perform at most $O\left(\frac{1}{\varepsilon}(H_0+1)\log n\right)$ steps going backward in T' (via the LF-mapping over $\mathcal{A}'$), position by position, until reaching

Algorithm Huff-FM_Position(i,B,Bh,ST)
(1) $d = 0$;
(2) **while** $S[rank(Bh, i)] = 0$ **do**
(3) **do if** $B[i] = 0$ then $i = i - rank(B, i) + [i < p_{\#}]$;
else $i = n' - rank(B, n') + rank(B, i)$;
(4) **while** $Bh[i] = 0$;
(5) $d = d + 1$;
(6) **return** $d + ST[rank(S, rank(Bh, i))]$;

Figure 3: Algorithm for reporting the text position of the occurrence at $B[i]$. It is invoked for each $i = select(Bh, r + k)$, $1 \leq k \leq occ$, $r = rank(Bh, sp - 1)$.

Algorithm Huff-FM_Display(l,r,B,Bh,TS)
(1) $j = \min\{k,\ ST[rank(S, rank(Bh, TS[k]))] > r\}$; // binary search
(2) $i = TS[j]$;
(3) $p = ST[rank(S, rank(Bh, i))]$;
(4) $L = \langle\ \rangle$;
(5) **while** $p \geq l$ **do**
(6) **do** $L = B[i] \cdot L$;
(7) **if** $B[i] = 0$ **then** $i = i - rank(B, i) + [i < p_{\#}]$;
else $i = n' - rank(B, n') + rank(B, i)$;
(8) **while** $Bh[i] = 0$;
(9) $p = p - 1$;
(10) Huffman-decode the first $r - l + 1$ characters from list L;

Figure 4: Algorithm for extracting $T[l \ldots r]$.

the first bit of the codeword for $T[r + 1]$. Then, we obtain the L preceding positions of T, by further traversing T' backwards, collecting all its bits until reaching the first bit of the codeword for $T[l]$. The reversed bit stream collected is Huffman-decoded to obtain $T[l \ldots r]$.

Each of those L characters costs us $O(H_0 + 1)$ on average because we obtain the codeword bits one by one. In the worst case they cost us $O(\log n)$. The overall time complexity is $O((H_0 + 1)(L + \frac{1}{\varepsilon} \log n))$ on average and $O(L \log n + (H_0 + 1)\frac{1}{\varepsilon} \log n)$ in the worst case. Figure 4 shows the pseudocode.

5 K-ary Huffman

The purpose of the idea of compressing the text before constructing the index is to remove the sharp dependence of the alphabet size of the FM index. This compression is done using a binary alphabet. In general, we can use Huffman over a coding alphabet of $k > 2$ symbols and use $\lceil \log k \rceil$ bits to represent each symbol. Varying the value of k yields interesting time/space tradeoffs. We use only powers of 2 for k values, so each symbol can be represented without wasting space.

The space usage varies in different aspects. Array B increases its size since the compression ratio gets worse. B has length $n' < (H_0^{(k)} + 1)n$ symbols, where $H_0^{(k)}$ is the zero order entropy of the text computed using base k logarithm, that is, $H_0^{(k)} =$

$-\sum_{i=1}^{\sigma} \frac{n_i}{n} \log_k \left(\frac{n_i}{n}\right) = H_0 / \log_2 k$. Therefore, the size of B is bounded by $n' \log k = (H_0 + \log k)n$ bits. The size of Bh is reduced since it needs one bit per symbol, and hence its size is n'. The total space used by these structures is then $n'(1 + \log k) < n(H_0^{(k)} + 1)(1 + \log k)$, which is not larger than the space requirement of the binary version, $2n(H_0 + 1)$, for $1 \leq \log k \leq H_0$.

The *rank* structures also change their size. The *rank* structures for Bh are computed in the same way of the binary version, and therefore they reduce their size, using $o(H_0^{(k)} n)$ bits. For B, we can no longer use the *rank* function to simulate Occ. Instead, we need to calculate the occurrences of each of the k symbols in B. For this sake, we precalculate sublinear structures for each of the symbols, including k tables that count the occurrences of each symbol in a chunk of $b = \lceil \log_k(n)/2 \rceil$ symbols. Hence, we need $o(kH_0^{(k)} n)$ bits for this structures. In total, we need $n(H_0^{(k)} + 1)(1 + \log k) + o(H_0^{(k)} n(k + 1))$ bits.

Regarding the time complexities, the pattern has length $< m(H_0^{(k)} + 1)$ symbols, so this is the search complexity, which is reduced as we increase k. For reporting queries and displaying text, we need the same additional structures TS, ST and S that for the binary version. The k-ary version can report the position of an occurrence in $O\left(\frac{1}{\epsilon}(H_0^{(k)} + 1) \log n\right)$ time, which is the maximum distance between two sampled positions. Similarly, the time to display a substring of length L becomes $O((H_0^{(k)} + 1)(L + \frac{1}{\epsilon} \log n))$ on average and $O(L \log n + (H_0^{(k)} + 1)\frac{1}{\epsilon} \log n)$ in the worst case.

6 Experimental Results

In this section we show experimental results on counting, reporting and displaying queries and compare the efficiency to existing indexes. The indexes used for the experiments were the FM-index implemented by Navarro [18], Sadakane's CSA [19], the RLFM index [17], the SSA index [17], and the LZ index [18]. Other indexes, like the Compressed Compact Suffix Array (CCSA) of Mäkinen and Navarro [16], the Compact SA of Mäkinen [15] and the implementation of Ferragina and Manzini of the FM-index were not included because they are not comparable to the FM Huffman index due either to their large space requirement (Compact SA) or their high search times (CCSA and original FM index).

We considered three types of text for the experiments: 80 MB of English text obtained from the TREC-3 collection [3] (files `WSJ87-89`), 60 MB of DNA and 55 MB of protein sequences, both obtained from the BLAST database of the NCBI[4] (files `month.est_others` and `swissprot` respectively).

Our experiments were run on an Intel(R) Xeon(TM) processor at 3.06 GHz, 2 GB of RAM and 512 KB cache, running Gentoo Linux 2.6.10. We compiled the code with `gcc 3.4.2` using optimization option `-O9`.

Now we show the results regarding the space used by our index and later the results of the experiments classified by query type.

[3] Text Retrieval Conference, `http://trec.nist.gov`

[4] National Center for Biotechnology Information, `http://www.ncbi.nlm.nih.gov`

6.1 Space results

Table 1 (left) shows the space that the index takes as a fraction of the text for different values of k and the three types of files considered. These values do not include the space required to report positions and display text.

We can see that the space requirements are lowest for $k = 4$. For higher values this space increases, although staying reasonable until $k = 16$. With higher values the spaces are too high for these indexes to be comparable to the rest.

We did not consider the version of the index with $k = 8$ in the other experiments because we do not expect an improvement in the query time, since $\log k$ is not a power of 2 and then the computation of Occ is slower (reasons omitted for lack of space). The version with $k = 16$ can lead to a reduction in query time, but the access to 4 machine words for the calculation of Occ (reasons omitted for lack of space) could negatively affect it. It is important to say that these values are only relevant for the English text and proteins, since it does not make sense to use them for DNA.

It is also interesting to see how the space requirement of the index is divided among its different structures. Table 1 (right) shows the space used by each of the structures for the index with $k = 2$ and $k = 4$ for the three types of texts considered.

k	Fraction of text		
	English	DNA	Proteins
2	1,68	0,76	1,45
4	1,52	0,74	1,30
8	1,60	0,91	1,43
16	1,84	—	1,57
32	2,67	—	1,92
64	3,96	—	—

	FM-Huffman $k = 2$			**FM-Huffman** $k = 4$		
Structure	Space [MB]			Space [MB]		
	English	DNA	Proteins	English	DNA	Proteins
B	48.98	16.59	29.27	49.81	18.17	29.60
Bh	48.98	16.59	29.27	24.91	9.09	14,80
$Rank(B)$	18,37	6,22	10,97	37,36	13,63	22,20
$Rank(Bh)$	18,37	6,22	10,97	9,34	3,41	5,55
Total	134,69	45,61	80,48	121,41	44,30	72,15
Text	80,00	60,00	55,53	80,00	60,00	55,53
Fraction	1.68	0.76	1.45	1.52	0.74	1.30

Table 1: On top, space requirement of our index for different values of k. The value corresponding to the row $k = 8$ for DNA actually corresponds to $k = 5$, since this is the total number of symbols to code in this file. Similarly, the value of row $k = 32$ for the protein sequence corresponds to $k = 24$. On the bottom, detailed comparison of $k = 2$ versus $k = 4$. We omit the the spaces used by the Huffman table, the constant-size tables for $Rank$, and array C, since they are negligible.

For higher values of k the space used by B will increase since the use of more symbols for the Huffman codes increases the resulting space. On the other hand, the size of Bh decreases at a rate of $\log k$ and so do its *rank* structures. However, the space of the *rank* structures of B increases rapidly, as we need k structures for an array that reduces its size at a rate of $\log k$, which is the reason of the large space requirement for high values of k.

6.2 Counting queries

For the three files, we show the search time as a function of the pattern length, varying from 10 to 100, with a step of 10. For each length we used 1000 patterns taken from random positions of each text. Each search was repeated 1000 times. Figure 5 (left) shows the time for counting the occurrences for each index and for the three files considered. As the CSA index needs a parameter to determine its space for this type of queries, we adjusted it so that it would use approximately the same space of the binary FM-Huffman index.

We show in Figure 5 (right) the average search time per character along with the minimum space requirement of each index to count occurrences. Unlike the CSA, the other indexes do not need a parameter to specify their size for counting queries. Therefore, we show a point as the value of the space used by the index and its search time. For the CSA index we show a line to resemble the space-time tradeoff for counting queries.

6.3 Reporting queries

We measured the time that each index took to search for a pattern and report the positions of the occurrences found. From the English text and the DNA sequence we took 1000 random patterns of length 10. From the protein sequence we used patterns of length 5. We measured the time per occurrence reported varying the space requirement for every index except the LZ, which has a fixed size. For the CSA we set the two parameters, namely the size of the structures to report and the structures to count, to the same value, since this turns out to be optimal. Figure 6 (left) shows the times per occurrence reported for each index as a function of its size.

6.4 Displaying text

We measured the time to display a context per character displayed. That is, we searched for the 1000 patterns and displayed 100 characters around each of the positions of the occurrences found. Figure 6 (right) shows this time along with the minimum space required for each index for the counting functionality, since the display time per character does not depend on the size of the index. This is not true for the CSA index, whose display time does depend on its size. For this index we show the time measured as a function of its size.

6.5 Analysis of Results

We can see that our FM-Huffman $k = 16$ index is the fastest for counting queries for English and proteins and that the version with $k = 4$ is, together with the SSA,

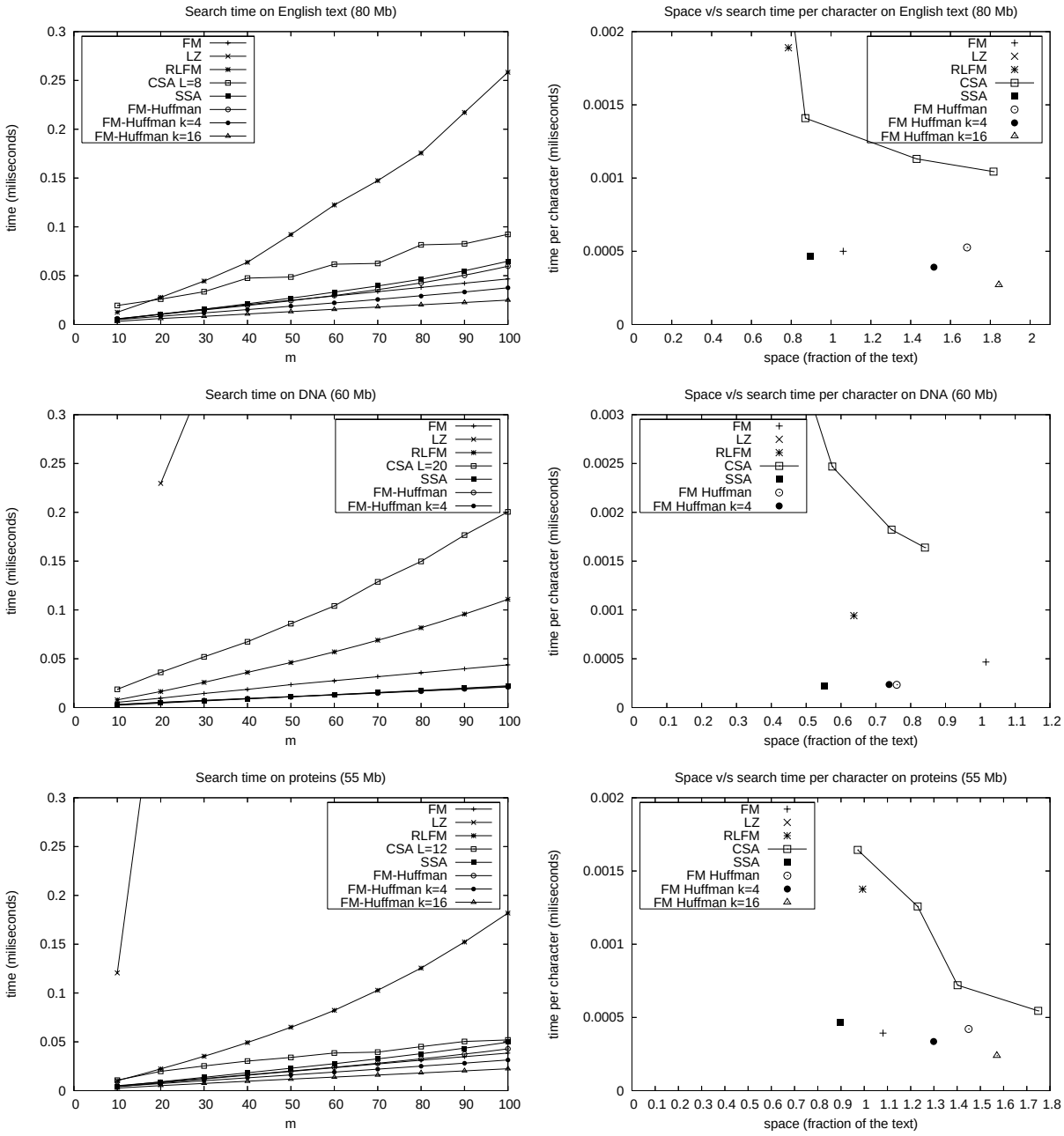


Figure 5: On the left, search time as a function of the pattern length over, English (80 MB), DNA (60 MB), and a proteins (55 MB). The times of the LZ index do not appear on the English text plot, as they range from 0.5 to 4.6 ms. In the DNA plot, the time of the LZ index for $m = 10$ is 0.26. The reason of this increase is the large number of occurrences of these patterns, which influences the counting time for this index. On the right, average search time per character as a function of the size of the index.

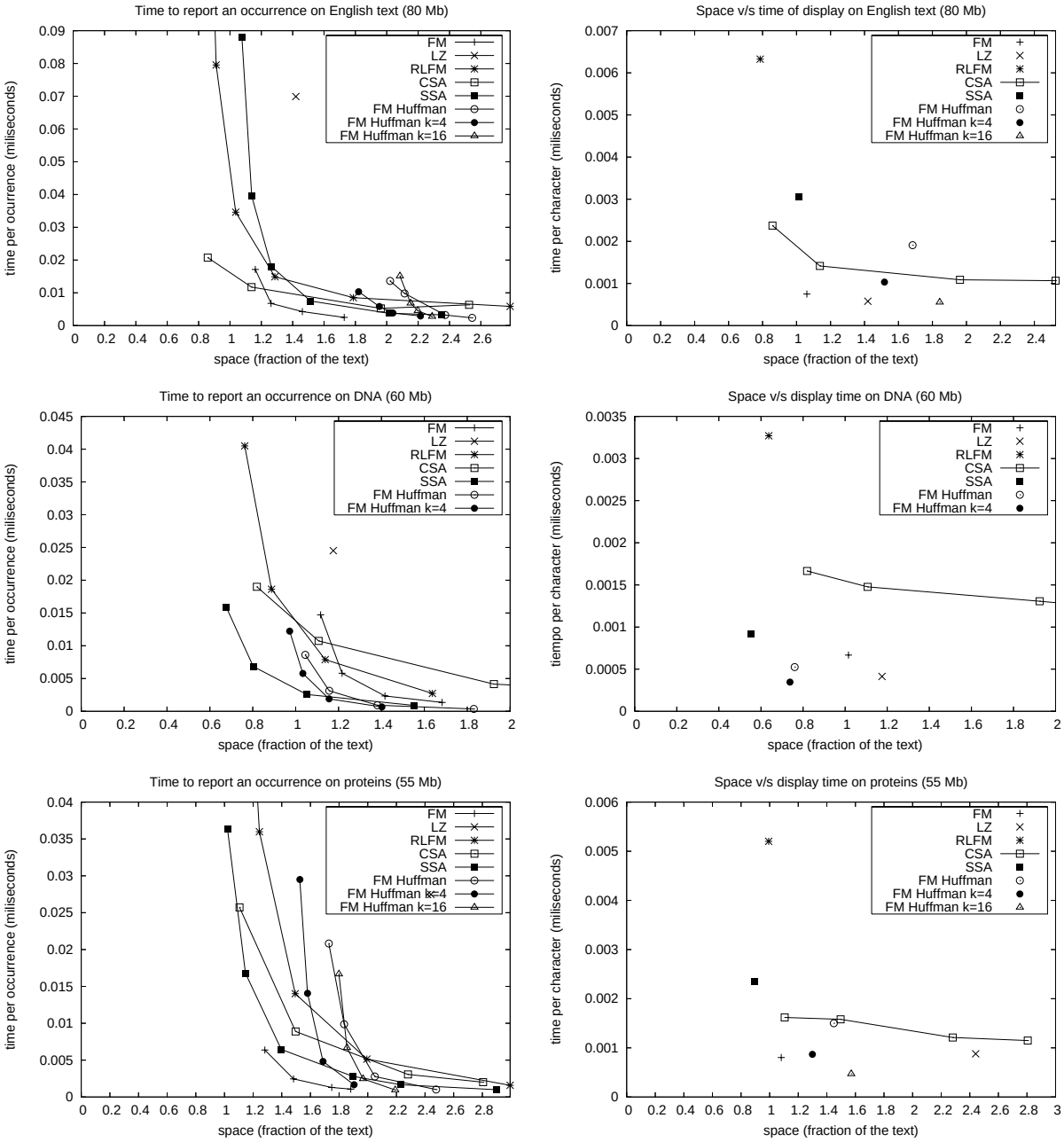


Figure 6: On the left, time to report the positions of the occurrences as a function of the size of the index. On the right, time per character to display text passages. We show the results of searching on 80 MB of English text, 60 MB of DNA and finally 55 MB of proteins.

the fastest for DNA. The binary FM-Huffman index takes the same time that $k = 4$ version for DNA and it is a little bit slower that the FM-index for the other two files. As expected, the three versions are faster than CSA, RLFM and LZ, the latter not being competitive for counting queries. Regarding the space usage, the SSA is an attractive tradeoff alternative for the three files, since it uses less space than our index and has low search times (although not as good as our index except on DNA). The same happens with the FM-index, although not for DNA, where it uses more space and time than our index.

For reporting queries, our index loses to the FM-index for English and proteins, mainly because of its large space requirement. Also, it only surpasses the RLFM and CSA, and barely the SSA, for large space usages. For DNA, however, our index, with $k = 2$ and $k = 4$, is better than the FM-index, although it loses to the SSA for low space usage. This reduction in space in our index is due to the low zero-order entropy of the DNA, which makes our index compact and fast.

Regarding display time, our index variants are again the fastest. On English text, however, the LZ is equally fast and smaller (version $k = 16$ is the relevant one here). On DNA, the $k = 4$ version is faster than any other, requiring also little space. Those taking (at best 20%) less space are about 3 times slower. Finally, on proteins, the version $k = 16$ is clearly the fastest. The best competitor, the FM-index, uses 30% less space but it is twice as slow.

The versions of our index with $k = 4$ improved the space and time of the binary version. The version with $k = 16$ increased the space usage, but resulted in the fastest of the three for counting and display queries. In general, our index is not the smallest but it is the fastest among those using the same space.

7 Conclusions

We have focused in this paper on a practical data structure inspired by the FM-index [3], which removes its sharp dependence on the alphabet size σ. Our key idea is to Huffman-compress the text before applying the Burrows-Wheeler transform over it. Over a text of n characters, our structure needs $O(n(H_0+1))$ bits, being H_0 the zero-order entropy of the text. It can search for a pattern of length m in $O(m(H_0+1))$ average time. Our structure has the advantage over the FM-index of not depending at all on the alphabet size, and of having better complexities to report text occurrences and displaying text substrings. In comparison to the CSA [19], it has the advantage of having better search time.

Furthermore, our structure is simple and easy to implement. Our experimental results show that our index is competitive in practice against other implemented alternatives. In most cases it is not the most succinct, but it is the fastest, even if we let the other structures use significatively more space.

References

[1] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. *DEC SRC Research Report 124*, 1994.

[2] D. Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, 1996.

[3] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. FOCS'00*, pp. 390–398, 2000.

[4] P. Ferragina and G. Manzini. An experimental study of an opportunistic index. In *Proc. SODA'01*, pp. 269–278, 2001.

[5] Sz. Grabowski, V. Mäkinen, and G. Navarro. First Huffman, then Burrows-Wheeler: an alphabet-independent FM-index. In *Proc. SPIRE'04*, pp. 210–211, 2004. Poster.

[6] Sz. Grabowski, V. Mäkinen, and G. Navarro. First Huffman, then Burrows-Wheeler: an alphabet-independent FM-index. Technical Report TR/DCC-2004-4. Dept. of Computer Science, Univ. of Chile, July 2004. `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/huffbwt.ps.gz`.

[7] R. González, Sz. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Poster Proc. WEA'05*, pp. 27–38, 2005.

[8] R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. SODA'03*, pp. 841–850, 2003.

[9] R. Grossi, A. Gupta, and J. Vitter. When indexing equals compression: Experiments with compressing suffix arrays and applications. In *Proc. SODA'04*, 2004.

[10] R. Grossi and J. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proc. STOC'00*, pp. 397–406, 2000.

[11] G. Jacobson. *Succinct Static Data Structures*. PhD thesis, CMU-CS-89-112, Carnegie Mellon University, 1989.

[12] J. Kärkkäinen. *Repetition-Based Text Indexes*, PhD Thesis, Report A-1999-4, Department of Computer Science, University of Helsinki, Finland, 1999.

[13] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22, pp. 935–948, 1993.

[14] I. Munro. Tables. In *Proc. FSTTCS'96*, pp. 37–42, 1996.

[15] V. Mäkinen. Compact Suffix Array — A space-efficient full-text index. *Fundamenta Informaticae* 56(1-2), pp. 191–210, 2003.

[16] V. Mäkinen and G. Navarro. Compressed compact suffix arrays. In *Proc. CPM'04*, pp. 420–433. LNCS 3109, 2004.

[17] V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. In *Proc. CPM'05*, pp. 45–56. LNCS 3537, 2005.

[18] G. Navarro. Indexing text using the Ziv-Lempel trie. *J. of Discrete Algorithms* 2(1):87–114, 2004.

[19] K. Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *Proc. ISAAC'00*, LNCS 1969, pp. 410–421, 2000.

[20] E. Ukkonen. On-line construction of suffix-trees. *Algorithmica*, 14, pp. 249–260, 1995.

[21] I. Witten, A. Moffat and T. Bell. *Managing Gigabytes*. Morgan Kaufmann Publishers, New York, 1999. Second edition.