Agència de Gestió d'Ajuts Universitaris i de Recerca
AGAUR

Memòria justificativa de recerca de les convocatòries BCC, BE, BP, CTP-AIRE, INEFC i PIV

La memòria justificativa consta de les dues parts que venen a continuació:
1.- Dades bàsiques i resums
2.- Memòria del treball (informe científic)

Tots els camps són obligatoris

1.- Dades bàsiques i resums

Nom de la convocatòria

BP

Llegenda per a les convocatòries:

BCC	Convocatòria de beques per a joves membres de comunitats catalanes a l'exterior
BE	Beques per a estades per a la recerca fora de Catalunya
BP	Convocatòria d'ajuts postdoctorals dins del programa Beatriu de Pinós
CTP-AIRE	Ajuts per accions de cooperació en el marc de la comunitat de treball dels Pirineus. Ajuts de mobilitat de personal investigador.
INEFC	Beques predoctorals i de col·laboració, dins de l'àmbit de l'educació física i l'esport i les ciències aplicades a l'esport
PIV	Beques de recerca per a professors i investigadors visitants a Catalunya

Títol del projecte: ha de sintetitzar la temàtica científica del vostre document.
Algorithmic Aspects of Bio-Inspired String Operations

Dades de l'investigador o beneficiari

Nom: Klaus-Peter
Cognoms: Leupold

Correu electrònic
klauspeter.leupold@urv.cat

Dades del centre d'origen

Departament de Filologies Romàniques
Universitat Rovira i Virgili
Tarragona

Número d'expedient
BPB0900052

Paraules clau: cal que esmenteu cinc conceptes que defineixin el contingut de la vostra memòria.
algorithm, string, bioinformatics, duplication, suffix array

Data de presentació de la justificació
19.11.2012

Nom i cognoms i signatura del/de la investigador/a

Vist i plau del/de la responsable de la sol·licitud

Generalitat de Catalunya
Departament d'Economia i Coneixement

Resum del projecte: cal adjuntar dos resums del document, l'un en anglès i l'altre en la llengua del document, on s'esmenti la durada de l'acció

Resum en la llengua del projecte (màxim 300 paraules)

We present building blocks for algorithms for the efficient reduction of square factor, i.e. direct repetitions in strings. So the basic problem is this: given a string, compute all strings that can be obtained by reducing factors of the form zz to z. Two types of algorithms are treated: an offline algorithm is one that can compute a data structure on the given string in advance before the actual search for the square begins; in contrast, online algorithms receive all input only at the time when a request is made.

For offline algorithms we treat the following problem: Let u and w be two strings such that w is obtained from u by reducing a square factor zz to only z. If we further are given the suffix table of u, how can we derive the suffix table for w without computing it from scratch? As the suffix table plays a key role in online algorithms for the detection of squares in a string, this derivation can make the iterated reduction of squares more efficient.

On the other hand, we also show how a suffix array, used for the offline detection of squares, can be adapted to the new string resulting from the deletion of a square. Because the deletion is a very local change, this adaption is more efficient than the computation of the new suffix array from scratch.

Resum en anglès (màxim 300 paraules)

We present building blocks for algorithms for the efficient reduction of square factor, i.e. direct repetitions in strings. So the basic problem is this: given a string, compute all strings that can be obtained by reducing factors of the form zz to z. Two types of algorithms are treated: an offline algorithm is one that can compute a data structure on the given string in advance before the actual search for the square begins; in contrast, online algorithms receive all input only at the time when a request is made.

For offline algorithms we treat the following problem: Let u and w be two strings such that w is obtained from u by reducing a square factor zz to only z. If we further are given the suffix table of u, how can we derive the suffix table for w without computing it from scratch? As the suffix table plays a key role in online algorithms for the detection of squares in a string, this derivation can make the iterated reduction of squares more efficient.

On the other hand, we also show how a suffix array, used for the offline detection of squares, can be adapted to the new string resulting from the deletion of a square. Because the deletion is a very local change, this adaption is more efficient than the computation of the new suffix array from scratch.

Memòria del Treball
Algorithmic Aspects of Bio-Inspired String Operations

Peter Leupold
Bestriu de Pinós Fellow BPB0900052 at the
Research Group in Mathematical Linguistics
Rovira i Virgili University, Tarragona, Spain
from November 2010 till October 2012

0 Introduction

A mutation which occurs frequently in DNA strands is the duplication of a factor inside a strand [6]. The result is called a tandem repeat, and the detection of these repeats has received a great deal of attention in bioinformatics [1, 10]. The reconstruction of possible duplication histories of a gene is used for the construction of a phylogenetic network in the investigation of the evolution of a species [11]. Thus duplicating factors and deleting halves of squares is an interesting algorithmic problem with some motivation from bioinformatics. A very similar reduction was also introduced in the context of data compression by Ilie et al. [4, 5]. They, however conserve information about each reduction step in the resulting string such that the operation can also be undone again. In this way the original word can always be reconstructed, which is essential for data compression.

Our main aim here is the development of efficient methods for the repeated reduction of squares. At the heart of this is the detection of squares, or, as we will see, the detection of runs. Several methods for this are known [7]. Usually they use suffix arrays or related data structures. What we want to avoid here is having to construct these for every string from scratch. Since the deletion of half a square is a very local change, it might be more efficient to update the old suffix array.

In recent work Salson et al. [9, 8] have investigated the updating of suffix arrays and related data structures. They considered insertions, deletions and changes of factors. Basically, the reduction of a square is just a deletion. However, it has the special property that another copy of the deleted factor remains just next to the deletion site. Thus the suffixes and LCP values of the new string's suffix array are more related to the old one's than usually. Here our aim is to characterize this relation and use it for an efficient update of the suffix array.

1 Runs, not Squares

Before we start to reduce squares, let us take a look at the effect that this operation has in periodic factors. In the following example, we see that reduction of either of the three squares

in the periodic factor *bcbcbc* leads to the same result:

```
abcbcbca        abcbcba        abcbcbca
            ↘      ↓      ↙
                abcbca
```

Thus it would not be efficient to do all the three reductions. A maximal periodic factor like this is called a *run*. So rather than looking for squares, we should actually look for runs and reduce one square within each of them.

As stated above, the most common algorithms for detecting runs are based on suffix arrays and related data structures [7]. Using these, we would employ a method along the lines of Algorithm 1. Then this method would be applied to all the resulting strings which are not

Algorithm 1: Constructing all words reachable from w by reduction of squares.

```
   Input: string: w;
   Data: stringlist: S (contains w);
1  while (S nonempty) do
2      x := POP(S);
3      Construct the suffix array of x;
4      if (there are runs in x) then
5          foreach run r do
6              Reduce r;
7              Add new string to S;
8          end
9      end
10     else output x;
11 end
```

square-free. Our aim is to improve line 3 by modifying the antecedent suffix array instead of constructing the new one from scratch. For this we first look at what a suffix array is.

SA	LCP	
7	1	a
0	0	abcbbcba
6	1	ba
3	1	bbcba
5	3	bcba
1	0	bcbbcba
4	2	cba
2		cbbcba

$\Rightarrow$

SA	LCP	
$7-3=4$	1	a
0	0	(new) abcba
$6-3=3$	1	ba
		—
$5-3=2$	1	bcba
		—
$4-3=1$		cba
		—

Figure 1: Modification of the suffix array by deletion of bcb in abcbbcba.

2 Suffix Arrays

In string algorithms, suffix arrays are a very common data structure, because they allow fast search for patterns. A suffix array of a string w consists of the two tables depicted on the left-hand side of Figure 1: SA is the lexicographically ordered list of all the suffixes of w; typically their starting position is saved rather than the entire suffix. LCP is the list of the longest common prefixes between these suffixes. Here we only provide the values for direct neighbors. Depending on the application, they may be saved for all pairs.

On the right-hand side of Figure 1 we see how the deletion of `bcb` changes the suffix array. Obviously there is no change in the relative order nor in the LCP values for all the suffixes that start to the right of the deletion site; here it is more convenient to consider the first half of the square as the deleted one, because then we see immediately that also for the positions in the remaining right half nothing changes.

The only new suffix is `abcba`. It starts with the same letter as `abcbbcba`, the one it comes from; also the following *bcb* is the same as before, because the deleted factor is replaced by another copy of itself — only after that there can be change. Thus the new suffix will not be very far from the old one in lexicographic order. Formulating these observations in a more general and exact way will be the objective of the next section.

3 Updating the Suffix Array

The problem we treat here is the following: Given a string w with a square of length n starting at position k and given the suffix array of w, compute the suffix array of $w[0 \dots k-1]w[k+n \dots |w|-1]$. So $w[k-1 \dots k+n-1]$ is deleted, not $w[k+n \dots k+2n-1]$.

First we formulate the obvious fact that the positions to the right of a deleted square remain in the same order.

Lemma 1. *The lexicographic order of the suffixes of a string w and their longest common prefixes are the same as for the corresponding suffixes in a longer string uw.*

For updating a suffix array, this means that can simply copy the values for these. The positions to the left of the deleted site may change. We formulate the conditions for this in terms of the old suffix array values.

Lemma 2. *Let the LCP of two strings z and uvw be k and let $z < uvw$. Then z and $uvvw$ have the same LCP and $z < uvvw$ unless $LCP(z, uvw) \geq |uv|$; in the latter case also $LCP(z, uvvw) \geq |uv|$.*

This characterizes the conditions under which actually a change in the suffix array has to be done. Salson et al. have shown efficient ways for reordering a suffix array after a deletion [9]. So we do not enter into details about this here. Algorithm 2 implements the updating of a suffix array after the deletion of a square avoiding unnecessary work according to the observations of this section.

The test in line 2 checks exactly the condition of Lemma 2. Note that if $LCP(u, v) < k$ then $LCP(wu, wv) < k + |w|$; thus as soon as the test fails once, we do not need to continue testing for longer suffixes. Rather we can stop the updating immediately, because the following LCP values will all fail the test.

The runtime of this updating depends very much on how often this test is successful. This, in turn, depends mainly on two factors: the length of the square that is reduced and the LCP values. The latter are higher for longer strings, because the probability of a factor occurring

Algorithm 2: Computing the new suffix array.

Input: string: w, SA, LCP;
length and pos of square: n,k;

1 $i := k - 1$;
2 **while** $(LCP[i] > n + k - i \; AND \; i \geq 0)$ **do**
3 | compute new SA of $w[i \ldots k-1]w[k+n \ldots |w|-1]$;
4 | compute new LCP[i];
5 | $i := i - 1$;
6 **end**

twice increases with the string's length; on the other hand, a larger alphabet decreases this probability. Both factors are not very much under our influence.

On the other hand, we can possibly do something about the length of the squares that are reduced. Squares of lengths one can be reduced first, if we do not want the entire reduction graph, but only the duplication root. For detecting and reducing them, it is faster to just run a window of size two over the string in low linear time without building the suffix array. After this, the value $n + k - i$ from line 2 of the algorithm would always be at least two. Squares of length two can already overlap with others in a way that reduction of one square makes reduction of the other impossible like in the string `abcbabcbc`; here reduction of the final `bcbc` leads to a square-free string, and the other root `abc` cannot be reached anymore.

Comparing theoretical worst case runtime, we have not achieved anything. There are algorithms for constructing suffix arrays in linear time. Salson et al.'s dynamic suffix arrays allow deletion in linear time, but in practice have proven much faster than the construction of a new suffix array. Similarly, our method will require linear time in the worst case. But as we have argued, the test in line 2 will often fail even in the first iteration. Then the computation consists only in removing the entries for the deleted positions. How much time this saves in practice can only be shown by experiments on large texts.

4 Online Detection of Squares

Now we focus on the detection of repetitions based on the so-called f-factorization of a string [2].

Definition 3. The f-factorization of a string w is the factorization $w = w_1 w_2 \cdots w_k$ such that every w_i is

- a letter that does not occur in $w_1 w_2 \cdots w_{i-1}$ or otherwise
- the longest prefix that occurs at least twice in $w_1 w_2 \cdots w_i$.

This factorization is a variant of the well-known Lempel-Ziv-factorization, where in the second clause "prefix" replaces "factor". For example, for the string `abaababa` we get the factorization

$$\mathtt{a} \cdot \mathtt{b} \cdot \mathtt{a} \cdot \mathtt{aba} \cdot \mathtt{ba}.$$

A key element in the computation of the f-factorization is the suffix table of the string. This table is defined as follows:

$$\mathsf{suff}[i] := |\mathsf{lcsuff}(w, w[0 \ldots i])|.$$

Here lcsuff is the function that computes the length of the longest common suffix of two strings. Thus the table consists of the lengths of the longest common suffixes between the string w and all of its prefixes. An example for a suffix table can be seen in Table 1.

Notice that the values are in general not very high, in our case between 0 and 3 with 0 being the most frequent one. In general, if all of the letters occur with equal probability, then the average value will be $\frac{1}{|\Sigma|} + \frac{1}{|\Sigma|^2} + \frac{1}{|\Sigma|^3} \cdots = \frac{1}{|\Sigma|-1}$, which will never be more than one.

In the example, the maximum of 3 is reached for the suffix `bab` of both `bbab` and the full string; on the other hand, half of the values are zero.

The computation of the suffix table by standard methods takes time linear in the size of the string.

Now we ask how much this table changes, when half of a square factor is deleted. If these changes are not fundamental, then we could use them for deriving the suffix table of the new string directly instead of computing it from scratch.

bbabcabab	suff
bbabcaba	0
bbabcab	2
bbabca	0
bbabc	0
bbab	3
bba	0
bb	1
b	1

Table 1: Suffix list for the string `bbabcabab`.

5 Modifying the Suffix Table

In the new string we distinguish three types of positions to analyze the changes in the suffix table:

$$\underbrace{\texttt{abbcabsab}}_{\text{left rest}}\texttt{baaaabb}\ \underbrace{\underbrace{\texttt{baaaabb}}_{\text{half square}}\ \underbrace{\texttt{cbbaccba}}_{\text{right rest}}}$$

The deleted positions obviously disappear and thus do not form part of the new string. Thus they do not show up in the suffix list either.

The positions in the right rest have the same suffixes as before; further, the corresponding suffixes of the entire string are the same, too. Thus all of these positions keep their values.

Notice that for the half square we actually had the choice of either the left or the right half – either deletion leads to the same result. Choosing to delete the left half as we have done immediately lets us see that for the positions in the half square the same as for the ones in the right rest is true: their suffix table values remain unchanged.

For the left rest, if the common suffix spanned beyond the deleted factor, then the string must have a periodic suffix. In this case, also these positions keep their values.

old string:	`cababababab`
suff[8]=8:	`cababababab`
new string:	`cababababab`
suff[8]=8:	`cababababab`

Figure 2: No change in the suffix list; deleted letters underlined.

However, if for a position i the value suff$[i]$ was smaller or equal to the length of the right rest plus the half square, then this value might change:

In these cases, the new value must be computed with new letter comparisons.

old string:	abca<u>bc</u>bcabc	old string:	abbc<u>b</u>bc
suff[5]=5:	abcabc	suff[3]=3:	abbc
new string:	abcabcabc	new string:	abbcbc
suff[5]=6:	abcabc	suff[3]=2:	abbc

Figure 3: The values in the suffix list can increase or decrease; deleted letters underlined

6 The Algorithm

As we have seen, there is basically one condition to check in order to determine, whether an entry in the suffix table needs to be changed. Thus the implementation of the update is quite straight-forward.

Algorithm 3: Updating the suffix list.

Data: w: the new string;
n: the length of w;
$oldsuff$: the suffix table of the old string;
k: the start position of the deleted factor in the old string;
ℓ: the end position of the deleted factor in the old string;
Result: $suff$: the suffix table for w

```
1  for (i := k to n − (ℓ − k) − 1) do
2  |  suff[i] := oldsuff[i + ℓ − k];
3  end
4  for (i := 0 to k − 1) do
5  |  if (oldsuff[i] < n − ℓ) then suff[i] := oldsuff[i];
6  |  else
7  |  |  suff[i] := n − ℓ;
8  |  |  j := n − ℓ + 1;
9  |  |  while (j ≤ i AND w[i − j] = w[n − j]) do
10 |  |  |  suff[i] := suff[i] + 1;
11 |  |  |  j := j + 1;
12 |  |  end
13 |  end
14 end
```

Thus the only letter comparisons are done in the test at line 9. The number is one more than the increase in the value of $suff$. As we have seen in the examples, in general these values are rather small. As a consequence, we can assume that in most cases very few comparisons will be done. Another factor that affects the this number is the distance from the end of the string: the bigger this distance, the higher the probability of the test in line 5 to be successful; and then the $suff$ value does not change.

The exact number of comparisons depends on the data. Compared to the computation of $suff$ from scratch, which takes between n and $2n - 1$ letter comparisons, our algorithm clearly improves the lower bound by reducing it to zero. Nonetheless, in the worst case this number might still be linear; It can be bounded to $2n - 1$ by similar techniques as for the original $suff$

construction: for every position at most one successful and one failing comparison are done. But this worst case is less probable and the lower bound is much better; thus updating is more efficient than computing from scratch.

7 Conclusion

As expected, the changes in suffix arrays and suffix tables that the deletion of squares cause are very local. We have shown how this can be taken advantage of for obtaining the suffix arrays and suffix tables of the resulting strings in a way that is more efficient than computing them from scratch.

References

[1] BENSON, D. A. Tandem Repeat Finder: A Program to Analyze DNA Sequences. *Nucleic Acids Research 27*, 2 (1999), 573–580.

[2] CROCHEMORE, M., HANCART, C., AND LECROQ, T. *Algorithms on Strings*. Cambridge University Press, New York, NY, USA, 2007.

[3] FREUND, R., HOLZER, M., TRUTHE, B., AND ULTES-NITSCHE, U., Eds. *Fourth Workshop on Non-Classical Models for Automata and Applications - NCMA 2012, Fribourg, Switzerland, August 23-24, 2012. Proceedings* (2012), vol. 290 of *books@ocg.at*, Österreichische Computer Gesellschaft.

[4] ILIE, L., YU, S., AND ZHANG, K. Repetition Complexity of Words. In: *COCOON* (2002), O. H. Ibarra and L. Zhang, Eds., vol. 2387 of *Lecture Notes in Computer Science*, Springer, pp. 320–329.

[5] ILIE, L., YU, S., AND ZHANG, K. Word Complexity And Repetitions In Words. *Int. J. Found. Comput. Sci. 15*, 1 (2004), 41–55.

[6] PENNISI, E. MOLECULAR EVOLUTION: Genome Duplications: The Stuff of Evolution? *Science 294*, 5551 (2001), 2458–2460.

[7] PUGLISI, S. J., SMYTH, W. F., AND YUSUFU, M. Fast, Practical Algorithms for Computing All the Repeats in a String. *Mathematics in Computer Science 3*, 4 (2010), 373–389.

[8] SALSON, M., LECROQ, T., LÉONARD, M., AND MOUCHARD, L. A four-stage algorithm for updating a Burrows-Wheeler transform. *Theor. Comput. Sci. 410*, 43 (2009), 4350–4359.

[9] SALSON, M., LECROQ, T., LÉONARD, M., AND MOUCHARD, L. Dynamic extended suffix arrays. *J. Discrete Algorithms 8*, 2 (2010), 241–257.

[10] SOKOL, D., BENSON, G., AND TOJEIRA, J. Tandem repeats over the edit distance. *Bioinformatics 23*, 2 (2007), 30–35.

[11] WAPINSKI, I., PFEFFER, A., FRIEDMAN, N., AND REGEV, A. Natural History and Evolutionary Principles of Gene Duplication in Fungi. *Nature 449* (2007), 54–61.

Publications during the Fellowship Period

[1] FAZEKAS, S. Z., LEUPOLD, P., AND SHIKISHIMA-TSUJI, K. Palindromes and Primitivity. In: *Automata and Formal Languages, 13th Intl. Conf, Proceedings* (2011), P. Dömösi and S. Iván, Eds., College of Nyíregyháza, pp. 184–196.

[2] HUNDESHAGEN, N., AND LEUPOLD, P. Transducing by Observing and Restarting Transducers. In Freund et al. [3], pp. 93–106.

[3] HUNDESHAGEN, N., AND LEUPOLD, P. Transducing by Observing and Restarting Transducers. *Invited to a special issue of Information and Computation* . (2012), .

[4] KRASSOVITSKIY, A., AND LEUPOLD, P. Computing by Observing Insertion. In: *LATA* (2012), A. H. Dediu and C. Martín-Vide, Eds., vol. 7183 of *Lecture Notes in Computer Science*, Springer, pp. 377–388.

[5] LEUPOLD, P. Reducing Repetitions. In: *Proceedings Workshop Algorithmic and Computational Theory in Algebra and Languages* (Research Institute for Mathematical Sciences, Kyoto, 2011), A. Yamamura, Ed., Kokyuruko Series.

[6] LEUPOLD, P. Computing by Observing Change in Insertion/Deletion Systems. In: Freund et al. *Fourth Workshop on Non-Classical Models for Automata and Applications - NCMA 2012, Fribourg, Switzerland, August 23-24, 2012. Proceedings* (2012), vol. 290 of *books@ocg.at*, Österreichische Computer Gesellschaft, pp. 123–132.

[7] LEUPOLD, P. Deletion of Squares in Suffix Arrays. In: *Proceedings Workshop Algebraic Systems and Theoretical Computer Science* (Research Institute for Mathematical Sciences, Kyoto, 2012), A. Yamamura, Ed., Kokyuruko Series.

[8] LEUPOLD, P. Efficient Reduction of Square Factors in Strings. In: *Proceedings LA Symposium* (Research Institute for Mathematical Sciences, Kyoto, 2012), Kazuo Iwama, Ed., Kokyuruko Series.

Congresses Attended

- Workshop Algorithmic and Computational Theory in Algebra and Languages, Kyoto, February 2011.
- LATA - 6th International Conference on Language and Automata Theory and Applications, A Coruña, Spain, March 5-9, 2012
- Language and Automata Symposium (Research Institute for Mathematical Sciences, Kyoto, February 2012.
- Workshop Algebraic Systems and Theoretical Computer Science, Kyoto, February 2012.
- Fourth Workshop on Non-Classical Models for Automata and Applications - NCMA 2012, Fribourg, Switzerland, August 23-24, 2012

Nom i cognoms i signatura del/de la investigador/a

Vist i plau del/de la responsable de la sol.licitud

Klaus-Peter Leupold