数理解析研究所講究録
第 1809 巻 2012 年 140-144

Reduction of Squares in Suffix Arrays

Peter Leupold*
Research Group in Mathematical Linguistics
Rovira i Virgili University, Tarragona, Spain
eMail: Peter.Leupold@web.de

0 Reducing Squares

A mutation which occurs frequently in DNA strands is the duplication of a factor inside a strand [5]. The result is called a tandem repeat, and the detection of these repeats has received a great deal of attention in bioinformatics [1, 9]. 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 [10]. Thus duplicating factors and deleting halves of squares is an interesting algorithmic problem with some motivation from bioinformatics, although squares do not always need to be exact there. A very similar reduction was also introduced in the context of data compression by Ilie et al. [2, 3]. 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 [6]. 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. [8, 7] 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 Repetitions and Duplication

We introduce a few formalisms to describe the reductions of squares in strings. We call a string w *square-free* iff it does not contain any non-empty factor of the form u^2, where exponents of strings refer to iterated catenation, and thus u^i is the i-fold catenation of the string u with itself. A string w has a positive integer

*Peter Leupold's work was done while he was funded as a Beatriu de Pinós researcher by the Departament d'Universitats, Recerca i Societat de la Informació de la Generalitat de Catalunya. His stay in Japan was kindly supported by Prof. Masami Ito from Kyoto Sangyo University

k as a *period*, if for all i, j such that $i \equiv j (\mathrm{mod} k)$ we have $w[i] = w[j]$, if both $w[i]$ and $w[j]$ are defined.

We formalize the *duplication relation* as a string-rewriting system defined as
$u \heartsuit v :\Leftrightarrow \exists z[z \in \Sigma^{+} \wedge u = u_1 z u_2 \wedge v = u_1 z z u_2]$. Notice how the symbol $\heartsuit$ nicely visualizes the operation going from one origin to two equal halves. $\heartsuit^{*}$ is the reflexive and transitive closure of the relation $\heartsuit$. The *duplication closure* of a string w is then $w^{\heartsuit} := \{u : w \heartsuit^{*} u\}$. Because our main topic is the reduction of squares, we will mainly use the inverse of $\heartsuit$ and will denote it by $\rightarrowtail := \heartsuit^{-1}$. The set $IRR(\rightarrowtail)$ of the strings irreducible under $\rightarrowtail$ is exactly the set of all square-free strings. With this we have all the prerequisites for defining the central notion of this work, the duplication root.

Definition 1. The *duplication root* of a non-empty word w is

$$\sqrt[\heartsuit]{w} := IRR(\rightarrowtail) \cap \{u : w \rightarrowtail^{*} u\}.$$

As usual, this notion is extended in the canonical way from words to languages such that

$$\sqrt[\heartsuit]{L} := \bigcup_{w \in L} \sqrt[\heartsuit]{w}.$$

If we do not only want the irreducible strings, then we use the notation
$w^{\rightarrowtail} := \{u : w \rightarrowtail^{*} u\}$.

When talking about squares, we will say that a square u^2 is of length $|u|$; in this case u will be called the *base* of this square.

Concerning the size of duplication roots we know that they can be exponentially large in terms of a string's length.

Theorem 2. *[4] The number duproots(n) of duplication roots of the string of length n with the maximum number of such roots satisfies*
$\frac{1}{30} 110^{\frac{n}{42}} \leq duproots(w) \leq 2^{n}$ *for all $n > 0$.*

2 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:

$$\texttt{a}\underline{\texttt{bcbc}}\texttt{bca} \qquad \texttt{ab}\underline{\texttt{cbcb}}\texttt{a} \qquad \texttt{abc}\underline{\texttt{bcbc}}\texttt{a}$$

$$\searrow \qquad \downarrow \qquad \swarrow$$

$$\texttt{abcbca}$$

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 [6]. Using these, we would employ a method along the lines of Algorithm 1. Then this method would

---

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**

---

be applied to all the resulting strings which are not 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			SA	LCP	
7	1	a		$7-3=4$	1	a
0	0	abcbbcba		0	0	(new) abcba
6	1	ba		$6-3=3$	1	ba
3	1	bbcba	$\Rightarrow$			—
5	3	bcba		$5-3=2$	1	bcba
1	0	bcbbcba				—
4	2	cba		$4-3=1$		cba
2		cbbcba				—

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

3 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.

4 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 $w[0 \ldots k-1]w[k+n \ldots |w|-1]$. So $w[k-1 \ldots k+n-1]$ is deleted, not $w[k+n \ldots k+2n-1]$.

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

Lemma 3. *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 4. *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 $LCP(z, uvvw) \geq |uv|$.*

Proof. If $LCP(z, uvw) < |uv|$ then the first position from the left where z and uvw differ is within uv. As uv is also a prefix of $uvvw$, z and $uvvw$ have their first difference in the same position. Thus LCP and the lexicographic order remain the same.

If $LCP(z, uvw) \geq |uv|$, then uv is a common prefix of z and $uvvw$. Thus 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 [8]. 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.

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**

The test in line 2 checks exactly the condition of Lemma 4. 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 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.

5 Perspectives

We have only looked at how to update a suffix array efficiently. But for actually computing a duplication history or a duplication root several more problems must be handled in an efficient way: as one word can produce many descendants, many suffix arrays must be derived from the same one and be stored; can this be done better than just storing them all in parallel? A typical duplication history contains many paths to a given word; how do we avoid computing a word more than once?

References

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

[2] 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.

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

[4] LEUPOLD, P. Reducing Repetitions. In: *Prague Stringology Conference* (Prague, 2009), J. Holub and J. Zdárek, Eds., Prague Stringology Club Publications, pp. 225–236.

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

[6] 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.

[7] 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.

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

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

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