Reducing Squares in Suffix Arrays

Peter Leupold

Institut für Informatik,
Universität Leipzig,
Leipzig, Germany
Peter.Leupold@web.de

Abstract. In contrast to other mutations, duplication leaves an easily detectable trace: a repetition. Therefore it is a convenient starting point for computing a phylogenetic network. Basically, all squares must be detected to compute all possible direct predecessors. To find all the possible, not necessarily direct predecessors, this process must be iterated for all the resulting strings. We show how to reuse the work for one string in this process for the detection of squares in all the strings that are derived from it. For the detection of squares we propose suffix arrays as data structure and show how they can be updated after the reduction of a square.

Keywords: duplication, suffix array

1 Introduction

The duplication of parts of a sequence is a very frequent gene mutation in DNA [5]. The result of such a duplication is called a tandem repeat. As duplications occur frequently in genomes, they can offer a way of reconstructing the way in which different populations of a given species have developed. For example, Wapinski et al. [8] wanted to determine the relations between seventeen different populations of a species of fungi. They did this by looking at the tandem repeats in the genes. From these they induced possible relationships.

In a very simplified way, the approach works like this: suppose at the same location in the genome of different individuals (or entire populations) of the same species we have the following sequences: uv, uuv, $uvuv$, and $uvuvvuv$. In this case, it is possible and even probable that the latter sequences have evolved from the first one via duplication. The second and third sequences are direct derivations from the first. To reach the last sequence, several duplications are necessary. So the question is, whether the corresponding individual can be a descendant of one or both of the others. Further duplications in uuv cannot change the fact that there are two successive letters u in the string. Since $uvuvvuv$ does not contain uu, it cannot be a descendant. On the other hand, it is obtained from $uvuv$ by duplicating the last three letters. So in this case the structure of repeats suggests a relationship like $uuv \Leftarrow uv \Rightarrow uvuv \Rightarrow uvuvvuv$. It is the only relationship that is possible using only duplication.

In general, the representation of evolutionary relationships between different nucleotide sequences, genes, chromosomes, genomes, or species is called a *phylogenetic network* [1]. Duplication is a mutation that is relatively easy to detect. It results in a repeat and thus leaves a visible and detectable trace unlike a deletion or a substitution. Thus possible predecessors can be computed by only looking at the sequence under question instead of comparing it to candidates for predecessors. In order to find possible ancestors via duplication, in principle the entire possible duplication history of a string must be reconstructed. Figure 1 graphically displays such a history. The main task here is finding repetitions in and thus possible ancestors of the given string.

Peter Leupold: Reducing Squares in Suffix Arrays, pp. 192–201.
Proceedings of PSC 2014, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05547-2

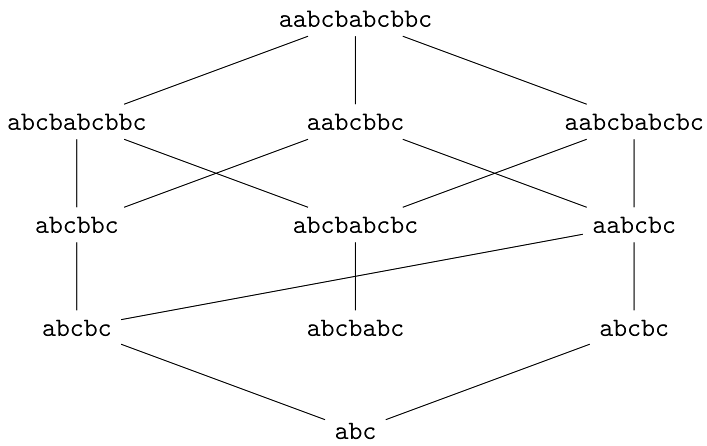

Figure 1. The duplication history for the string `aabcbabcbbc`. The direction of reductions is top to bottom.

There exist efficient algorithms for the detection of tandem repeats like the ones implemented in the `mreps` web interface [2]. These take into account biochemical conditions and do not necessarily search for perfect repetitions. For strings, the detection of perfect repetitions is frequently based on suffix arrays or similar data structures [6]. There are also good algorithms for the construction of these. However, if we want to search for more repetitions in the predecessors, we have to compute their suffix arrays from scratch. Since the change in the string is only local, the new suffix array will be very similar to the old one. We aim to characterize this similarity and to show how the new suffix array can be computed by modifying the old one rather than computing the new one from scratch. Here we address this problem for strings and perfect repetitions without regarding possible biochemical restrictions of tandem repeats in real DNA.

It is worth noting that such a computation might prove infeasible already due to the mere size of the solution. Note that at the end of each complete sequence of reduction there is a square-free string, i.e. one without any repetition. Obviously no further repetition can be reduced, if there is none by definition. When we consider the elimination of repetitions as a string-rewriting process, these square-free strings are the original string's normal forms. Even the number of these normal forms, which form only the end points in the paths of the duplication histories, can be exponential in the length of the string.

Theorem 1 ([4]). *For every positive integer n there are words of length n whose number N of normal forms under eliminating squares is bounded by:*

$$\frac{1}{30}110^{\frac{n}{42}} \leq N \leq 2^n.$$

Thus there can be no hope of finding an efficient algorithm for computing the entire duplication history for all possible strings. However, many strings will not reach this worst case. For them, we intend to find good methods for computing their duplication histories. The efficient detection of repetitions is a first step in such a computation.

2 Preliminaries on Strings

We recall some basic concepts on strings and fix notations for them. For a string w we denote by $w[i]$ the symbol at position i, where we start counting at 0. $w[i \dots j]$ denotes the substring starting at position i and ending at position j. A string w has a positive integer k as a *period*, if for all i, j such that $i \equiv j (\bmod k)$ we have $w[i] = w[j]$, if both $w[i]$ and $w[j]$ are defined. The length of w is always a trivial period of w. If a string has a non-trivial period then it is called *periodic*.

A string u is a *prefix* of w if there exists an $i \leq |w|$ such that $u = w[0 \dots i]$; if $i < |w|$, then the prefix is called *proper*. Suffixes are the corresponding concept reading from the back of the word to the front. The *longest common prefix* of two strings u and v is the prefix $lcp(u, v) = u[0 \dots i] = v[0 \dots i]$ such that either $u[i+1] \neq v[i+1]$ or at least one of the strings has length only $i + 1$. So the function *lcp* takes two strings as its arguments and returns the length of their longest common prefix.

The *lexicographical order* is defined as follows for strings u and v over an ordered alphabet: $u \leq v$ if u is a prefix of v, or if the two strings can be factorized as $u = wau'$ and $v = wbv'$ for some w and letters a and b such that $a < b$.

3 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:

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

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

$$\texttt{abcbca}$$

Thus it would not be efficient to do all the three reductions and produce three times the same string. A maximal periodic factor like this is called a *run*. Maximal here means that if we choose a longer factor that includes the current one, then this longer factor does not have the same period any more. In the string above, *bcbcbc* is a run. It has period two, but its extensions *abcbcbc* to the left and *bcbcbca* to the right do not have period two any more.

it is rather straight-forward to see how the example from above generalizes to arbitrary periodic factors. For the sake of completeness we give a formal proof of this fact.

Lemma 2. *Let w be a string with period k. Then any deletion of a factor of length k will result in the same string.*

Proof. Because the string w has period k, the deleted factor starts with a suffix and ends with a prefix of $w[0 \dots k-1]$. Thus it is of the form

$$w[i+1 \dots k-1]w[0 \dots i]$$

for some $i < k$. If it starts at position $\ell + 1$, then the string

$$w[0 \dots \ell]w[i+1 \dots k-1]w[0 \dots i]w[\ell+k+1 \dots |w|-1]$$

is converted to

$$w[0 \dots \ell]w[\ell+k+1 \dots |w|-1].$$

For a deletion at position ℓ, which is one step more to the left, we obtain

$$w[0 \dots \ell-1]w[i]w[\ell+k+1 \dots |w|-1].$$

Since $w[i] = w[\ell+k]$ this letter is equal to $w[\ell]$ due to the period k, and thus

$$w[0 \dots \ell] = w[0 \dots \ell-1]w[i],$$

and the two results are the same. In an analogous way the deleted factor can be moved to the right. The result is the same, also for several consecutive movements. □

So rather than looking for squares, we should actually look for runs and reduce only one square within each of them. Then all the resulting strings will be different from each other.

As stated above, the most common algorithms for detecting runs are based on suffix arrays and related data structures [6]. Using these methods, we would employ a strategy along the lines of Algorithm 1. Then this method would again be applied

Algorithm 1: Computing all the strings 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 one square in r;
7              Add new string to S;
8          end
9      end
10     else output x;
11     ;
12 end
```

to all the resulting strings which are not square-free. Our main aim is to improve line 3 by modifying the antecedent suffix array instead of constructing the new one from scratch. For this we first recall what a suffix array is.

4 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 2: 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. SA and LCP are also called an *extended* suffix array in contrast to SA alone.

SA	LCP			SA	LCP	
7	1	a		$7-3=4$	1	a
0	0	abcbbcba		0	0	abcba (new)
6	1	ba		$6-3=3$	1	ba
3	1	bbcba	$\Rightarrow$			—
4	3	bcba		$5-3=2$	0	bcba
1	0	bcbbcba				—
5	2	cba		$4-3=1$		cba
2		cbbcba				—

Figure 2. Modification of the suffix array by deletion of `bcb` in `abcbbcba`.

Now let there be a run with period k that contains at least one square uu starting in position i in a string; this means the run has at least length $2k$. Then the positions i and $i+k$ have an LCP of at least k and are very close to each other in the suffix array, because both suffixes start with u. This is why suffix arrays can be used to detect runs without looking at the actual string again once the array is computed.

On the right-hand side of Figure 2 we see how the deletion of `bcb` changes the suffix array. 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, see also Figure 3.

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 some 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.

5 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 from the original string, not $w[k+n\dots k+2n-1]$. The result is, of course the same; however, it is convenient for our considerations to suppose that it is the first half of the square that is deleted. In this way, it is a little bit easier to see, which suffixes of the original string are also suffixes of the new one.

Figure 3 illustrates the simple fact that the positions to the right of a deleted square remain in the same order and that this is also true for the positions in the second half of the square, which is not deleted.

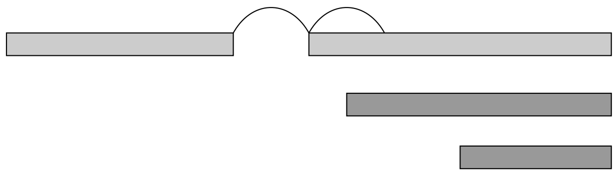

Figure 3. The order and LCP of suffixes that start right of the deletion remain unchanged.

For updating a suffix array, this means that only suffixes starting in positions to the left of the deleted site change and thus might also change their position in the suffix array. Figure 4 shows that no such change occurs if the LCP value is not greater than the sum of the length of the deleted factor (because this factor is replaced by another copy of itself from the right) and the distance from the start of the suffix to the start of the square (because it remains a prefix).

Only if the LCP is greater than this sum the suffix changes some of its first LCP many symbols as depicted in Figure 5. In this case we have to check if the position of the suffix in the suffix array changes. Lemma 3 formally proves these conditions.

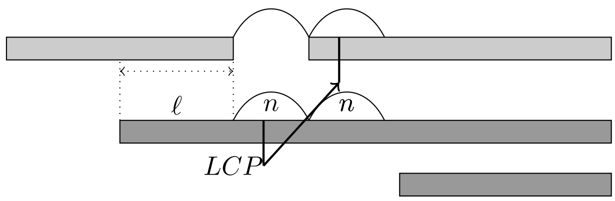

Figure 4. If the LCP is not greater than $\ell + n$, then the order of the new suffix relative to the old ones remains unchanged, because its $\ell + n$ are the same as before the deletion.

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

□

So we know that we only have to process suffixes starting to the left of the deleted factor. But also here not necessarily all suffixes have to be checked. To be more exact, as soon as starting from the right one suffix does not fulfill the conditions of Lemma 3, all the suffixes starting to the left of it will not fulfill these conditions either.

Lemma 4. *Let $LCP[j] = k$ in the suffix array of a string w of length $n + 1$. Then for $i < j$ we always have $LCP[i] \leq k + j - i$.*

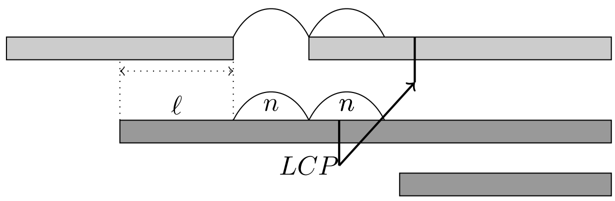

Figure 5. Only if the LCP is greater than $\ell + n$ some letter within the prefix of length LCP might change.

Proof. Let us suppose the contrary of the statement, i.e. $LCP[i] > k + j - i$. Further let the suffix of w that is lexicographically following $w[i \dots n]$ start at position m. This suffix shares a prefix of length at least $k + j - i + 1$ with $w[i \dots n]$, i.e.

$$w[m \dots m + k + 1] = w[i \dots i + k + 1].$$

Disregarding the first $j - i$ letters we obtain

$$w[m + (j - i) \dots m + (j - i) + k + 1] = w[i + (j - i) \dots i + (j - i) + k + 1]$$

and this gives us

$$w[m + j - i \dots m + j - i + k + 1] = w[j \dots j + k + 1].$$

So $w[i \dots n]$ shares $k + 1$ letters with the suffix $w[m + j - i \dots m + j - i + k + 1]$ of w. Further, since $w[m \dots n]$ is lexicographically greater than $w[i \dots n]$ also $w[m + j - i \dots n]$ is greater than $w[i \dots n]$. Therefore $LCP[i]$ must be at least $k+1$ contradicting our assumption.

□

Thus the best strategy seems to be to start at the first position left of the deleted factor and check the condition of Lemma 3. If it indicates that the position and LCP change, these changes are done and we move one position left. As soon as the condition of Lemma 3 says that no change can occur for the present position we can stop, because there will not be changes for the positions further to the left either.

Algorithm 2 implements this strategy avoiding unnecessary work according to the observations of this section. first we treat all the suffixes that start behind the deletion. They are simply decreased by n, the length of the deleted factor. The positions k to $k + n - 1$ are thus deleted. For the other suffixes we need to find out whether they must be moved to a different position. The test in line 5 checks exactly the condition of Lemma 3. Lemma 4 shows that after $LCP[i] > n+k-i$ we do not need to continue. The positions of the remaining suffixes can again be copied.

There is the variable m that appears in line 10 that is not bound. The reason is the following: If in line 6 a suffix is moved down in the array, then it is best to insert it directly in $SAnew$. If, on the other hand, it is moved up, then this part of $SAnew$ is not yet computed or copied. Therefore we should insert it into its position in the old SA and copy it in the final for-loop. So m should be the number of suffixes that are moved up in line 6 and it should be logged every time line 6 is executed.

It remains to implement the reordering and the computation of the new LCP value.

Algorithm 2: Computing the new suffix array.

Input: string: w; arrays: SA, LCP;
length and pos of square: n,k;
for $j = n + k$ **to** $|w| - 1$ **do**
  SAnew[j]:=SA[j]-n;
end
$i := k - 1$;
while $(LCP[i] > n + k - i \ AND \ i \geq 0)$ **do**
  compute SAnew of $w[i \dots k - 1]w[k + n \dots |w| - 1]$;
  compute new LCP[i];
  $i := i - 1$;
end
for $j = 0$ **to** $i + m$ **do**
  SAnew[j]:=SA[j];
end

6 Computing the Changes

There are two tasks whose details are left open in Algorithm 2. For those suffixes whose position changes, we need to find the new position. Then also all of the affected LCP-values must be updated.

6.1 Computing the New Position

Intuitively, the position of a new suffix is not very far from the position of the old suffix it is derived from. If a suffix $wuuv$ is converted to wuv, then the prefix wu remains the same. Both w and u are non-empty in our case, because only suffixes that start left of the deletion can change their position as Figure 3 showed. Therefore $|wu| \geq 2$ and consequently $lcp(wuuv, wuv) \geq 2$. Further, it is clear that also all the suffixes that are between the positions of $wuuv$ and wuv must start with wu and thus have a LCP with both of them that is greater or equal to $|wu|$. Therefore we can restrict our search for the position of wuv to the part of the suffix array around $wuuv$, where the LCP-values are not smaller than $|wu|$. Within this range we use the standard method for inserting a string in a suffix array.

6.2 Updating the LCP

At this point, where a suffix is moved to another position, the LCP-table must be updated to contain the new LCP between the predecessor and the successor. This is computed via the following, well-known equality: for three consecutive suffixes u, v, and w in the suffix array, always $lcp(u, w) = min(lcp(u, v), lcp(v, w))$ holds, see also the work of Salson et al. that treats deletions in general [7].

When a new suffix is inserted into the suffix array or moved to a new position, we actually need to compute two LCP values: the ones with either of the neighboring positions. Here we distinguish two cases.

If we have inserted at the position just after $i + \ell$ from the original position i, then we know that the LCP of wuv with the suffix at position $j + 1$ is at least $|wu|$ as explained in Section 6.1. Therefore we only need to test starting from the first letter after wu whether there are further matching letters.

The LCP with the following position might be as low as zero. In this case, however, also the original $LCP[i + \ell]$ was zero, because the two suffixes start with the same letter as they share the prefix $|wu|$. More generally, if $LCP[i + \ell]$ was less than $|wu|$, then the inserted suffix has the same LCP with its successor. So we only need to compare more letters if the old $LCP[i + \ell]$ is greater than $|wu|$. Otherwise we can inherit the old value.

For the case that the new position in the suffix array is higher, we proceed symmetrically. The LCP with the successor is at least $|wu|$, and the LCP with the predecessor can be taken from the original list unless this value was greater than $|wu|$.

7 Conclusion and Perspectives

In the best case our method will immediately detect that essentially no change in the suffix array need to be done. For long squares this is even probable, because the longer the square the smaller the probability that the LCP-value will be even bigger. Then the test in line 5 of Algorithm 2 immediately fails, and we essentially copy the relevant parts of the old suffix array.

On the other hand, even in the worst case we should save time compared to constructing the new suffix array from scratch. Everything behind the reduced square can be copied. And for the new suffixes that change position we can use some of the old information for making the finding of the new position and the computation of LCP easier. The real question is in how far we can do better than the general updating after a deletion that Salson et al. designed [7]. This can probably only be answered by testing actual implementations on data set in analyses like the ones carried out by Léonard et al. [3].

We have only looked at how to update a suffix array efficiently. But for actually computing a duplication history 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. Then all of these must be stored at the same time. Again, they are all similar to each other. The question is whether there are ways in which this similarity can be used to store them more compactly.

Further, a typical duplication history contains many paths to a given word. For example for a word $w_1u^2w_2v^2w_3x^2w_4$ that contains three squares that do not overlap, there is one normal form $w_1uw_2vw_3xw_4$ and there are six paths leading to this string. Every intermediate word is on two of these paths. The ones with one square left are reached from two different words, the normal form is reached from the three strings with only one square. How do we avoid computing a string more than once? Is there a way of knowing that the result was already obtained in an earlier reduction?

Depending in the goal of the computation, 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 normal forms. For detecting and reducing these squares, 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 5 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 normal form `abc` cannot be reached anymore.

References

1. D. H. Huson, R. Rupp, and C. Scornavacca: *Phylogenetic Networks*, Cambridge University Press, Cambridge, 2010.
2. R. Kolpakov, G. Bana, and G. Kucherov: *mreps: efficient and flexible detection of tandem repeats in DNA*. Nucleic Acids Research, 31(13) 2003, pp. 3672–3678.
3. M. Léonard, L. Mouchard, and M. Salson: *On the number of elements to reorder when updating a suffix array*. Journal of Discrete Algorithms, 11 2012, pp. 87–99.
4. P. Leupold: *Reducing repetitions*, in Prague Stringology Conference, J. Holub and J. Žďárek, eds., Prague Stringology Club Publications, Prague, 2009, pp. 225–236.
5. A. Meyer: *Molecular evolution: Duplication, duplication*. Nature, 421 2003, pp. 31–32.
6. S. J. Puglisi, W. F. Smyth, and M. Yusufu: *Fast, practical algorithms for computing all the repeats in a string*. Mathematics in Computer Science, 3(4) 2010, pp. 373–389.
7. M. Salson, T. Lecroq, M. Léonard, and L. Mouchard: *Dynamic extended suffix arrays*. Journal of Discrete Algorithms, 8(2) 2010, pp. 241–257.
8. I. Wapinski, A. Pfeffer, N. Friedman, and A. Regev: *Natural history and evolutionary principles of gene duplication in fungi*. Nature, 449 2007, pp. 54–61.