Minimum Unique Substrings and Maximum Repeats*

Lucian Ilie[1] and W. F. Smyth[2,3]

[1] Department of Computer Science, University of Western Ontario
London ON N6A 5B7, Canada
ilie@csd.uwo.ca
[2] Department of Computing and Software, McMaster University
Hamilton ON L8S 4K1, Canada
smyth@mcmaster.ca
[3] Digital Ecosystems and Business Intelligence Institute
Curtin University of Technology, Perth WA 6845, Australia

Abstract. Unique substrings appear scattered in the stringology literature and have important applications in bioinformatics. In this paper we initiate a study of *minimum unique substrings* in a given string; that is, substrings that occur exactly once while all their substrings are repeats. We discover a strong duality between minimum unique substrings and *maximum repeats* which, in particular, allows fast computation of one from the other. We give several optimal algorithms, some of which are very simple and efficient. Their combinatorial properties are investigated and a number of open problems are proposed.

1 Introduction

The contrasting ideas of *unique* and *repeating* substrings in a given string w are natural and intuitive: a unique substring occurs just once in w, a repeating substring more than once. Thus it is not surprising that these ideas have been explored in various ways by many authors in the past; what is perhaps surprising is that they have not been studied *together*, in such a way as to highlight — indeed to exploit — the contrast or duality between them. In this paper we take a first step toward the integrated study of unique and repeating substrings. We report results of combinatorial significance, we propose efficient new algorithms, and we propose applications and extensions of our methods, especially to computational biology.

Repeating substrings — specifically adjacent repeating substrings or "repetitions" — were the focus of what is acknowledged to be the paper that established combinatorics on words as a mathematical discipline [26]: more than a century ago, Thue showed how to construct infinitely long strings on a three-letter alphabet that contained no repetitions. Three-quarters of a century later, with the

* Both authors' research has been supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC).

advent of the electronic computer, three of the earliest and most influential papers in computational stringology [2, 7, 17] described algorithms for computing all the repetitions in a given string.

The discovery of the human genome sequence [5, 15] made the computation of nonadjacent repeats (copied DNA) an important algorithmic problem, one that became computationally feasible with the invention of the suffix tree by Weiner [27]: in the suffix tree T_w of w, repeating substrings could now be identified simply by locating the root of a corresponding subtree. Significantly, Weiner's paper also introduced the concept of a "minimum length unique substring" of w that begins at position i; that is, a substring $w[i\,..\,j]$ that occurs exactly once in w, while $w[i\,..\,j-1]$ occurs at least twice. However, no link was suggested between unique substrings and repeats.

In biological and other applications, not all repeats are of equal interest, and so methods were devised to use the suffix tree to compute repeats that were nonextendible to left or right [11, Chapter 7.12], that were separated by a gap of bounded size [4], that contained blocks of undefined (don't-care) symbols [8], or that occurred a bounded number of times in different segments of w [3]. With the discovery of suffix arrays in 1990 [18], and with the gradual recognition that their use saved time as well as space [1], many other algorithms were proposed for the computation of repeats [9, 13, 16, 19] that made use of these and other related data structures.

Meanwhile, in an apparently quite separate development, though using the same suffix tree/array techniques, many papers were written [10, 12, 24, 28] to find unique strings (oligonucleotides in the DNA context). In fact, the last of these cited papers is the only source we have found that gives the same definition of "minimum unique substring" that we give here: a substring that occurs exactly once in w and whose every proper substring occurs at least twice in w. Our contribution in this paper is to put together the ideas of "minimum unique" and "maximum repeat" — combinatorial properties arise from this duality that are interesting in their own right, and that also lead to new and efficient algorithms.

Section 2 of this paper gives basic definitions; then Section 3 introduces unique substrings and repeats, followed in Section 4 by an analysis of the duality between them. Section 5 presents linear-time algorithms to compute unique substrings and repeats using suffix arrays. Then Section 6 shows that de Bruijn strings are exactly those that maximize both the number of unique substrings and the number of repeats. Finally, Section 7 suggest possible future applications of these ideas.

2 Basic definitions

Let w be a string of *length* $|w| = n$. For $1 \le i \le n$, $w[i]$ is the ith letter of w. A *substring* of w is a string of the form $w[i\,..\,j] = w[i]w[i+1]\ldots w[j]$, for some $1 \le i \le j \le n$. In particular $w = w[1\,..\,n]$. The *reverse* of w is denoted $w_r = w[n]\ldots w[1]$. The *suffix* of w that starts at position i is denoted $\mathsf{suf}[i] = w[i\,..\,n]$.

The *suffix array* SA contains the positions $1, 2, \ldots, n$ sorted in increasing lexicographical order of the corresponding suffixes $\mathsf{suf}[i]$, $i = 1, 2, \ldots, n$. That is, $\mathsf{SA}[i] = j$ means that $\mathsf{suf}[j]$ is the ith smallest suffix in lexicographical order. The longest common prefix array, LCP, contains in its ith position the length of the longest common prefix of $\mathsf{suf}[\mathsf{SA}[i]]$ and $\mathsf{suf}[\mathsf{SA}[i-1]]$. An example is shown in Fig. 1.

As noted in Section 1, suffix arrays were introduced in 1990 by Manber and Myers [18], and today provide the most efficient text index. Currently the fastest suffix array construction algorithm (SACA) is due to [20], an algorithm that is linear in string length, fast in practice, and lightweight in its use of storage. See also [21] for a survey of SACAs and further analysis of their properties. The LCP array can also be computed in linear time and space [14]; what is at present the most economical algorithm is described in [23].

i	SA[i]	suf[SA[i]]	LCP[i]
1	8	a	
2	3	aababa	1
3	6	aba	1
4	1	abaababa	3
5	4	ababa	3
6	7	ba	0
7	2	baababa	2
8	5	baba	2

Fig. 1. The SA and LCP arrays for the string $w =$ abaababa. The third column shows the suffixes with the LCP positions underlined.

3 Unique substrings and repeats

A *repeat* of w is an interval $[i\,..\,j]$ such that the substring $w[i\,..\,j]$ occurs at least twice in w. A subinterval of a repeat is also a repeat and therefore it is natural to consider maximum repeats. A *maximum repeat* is an interval $[i\,..\,j]$ such that the string $w[i\,..\,j]$ occurs at least twice in w but the strings $w[i-1\,..\,j]$ and $w[i\,..\,j+1]$, if defined, do not. Maximum repeats include all repeats as subintervals.

A *unique substring* of w is an interval $[i\,..\,j]$ such that the substring $w[i\,..\,j]$ occurs exactly once in w. When computing unique substrings, it makes sense to compute only the minimum ones, as all the substrings of w containing those will be unique as well. That means, minimum ones determine all unique substrings. A *minimum unique substring* of w is an interval $[i\,..\,j]$ such that either $i = j$ and the letter $w[i]$ occurs only once in w or $i < j$ and each of the intervals $[i+1\,..\,j]$ and $[i\,..\,j-1]$, if defined, is a repeat of w.

As an example, the maximum repeats and minimum unique substrings for the string $w =$ abaababa are shown in Fig. 2.

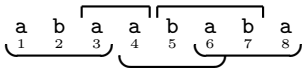

Fig. 2. The maximum repeats (half ovals: $[1\,..\,3], [4\,..\,6], [6\,..\,8]$) and minimum unique substrings (half rectangles: $[3\,..\,4], [5\,..\,7]$) of the string `abaababa`.

Notice that our definitions are in terms of intervals. For unique substrings, minimum or not, there is, conceptually, no difference if we define them as intervals or strings, because there is a bijection between the two concepts: an interval determines a string uniquely (which is always true) and a unique string determines an interval uniquely as well, as it has only one occurrence. The difference is important from an algorithmic point of view though, as it takes linear time to find the interval corresponding to a given string and only constant time vice versa.

Repeats are defined as intervals as well, with similar advantages. However, the correspondence between repeated strings and repeated intervals is more complex. Traditionally, "repeat" has usually been defined [1, 22, 25] in terms of the set of *all* occurrences of the repeating substring — in other words, all the intervals $[i'\,..\,i'+(j-i)]$ such that $w[i'\,..\,i'+(j-i)] = w[i\,..\,j]$. For example, in Fig. 2, ab is a repeating substring that occurs three times at positions 1, 4, and 6 with length 2, and so the repeat can be described by the tuple $(2;\ 1, 4, 6)$. In order to restrict repeats to those that are interesting, it is usual also to require that repeats be *nonextendible* (NE); that is, such that not every occurrence of a repeat is a substring of the same enclosing repeat. In our example, every occurrence of `ab` extends to the repeating substring `aba` and is therefore extendible; however, `aba` is nonextendible and so as an NE repeat is reported by $(3;\ 1, 4, 6)$.

In order to further restrict unnecessary output, it is common to report only repeats that are SNE or *supernonextendible* [22, 25] (also called "supermaximal" [1]) — that is, NE repeats whose repeating substring is not a proper substring of any other repeating substring of w. In Fig. 2 we see that the three maximum repeats collectively determine an SNE repeat $(3;\ 1, 4, 6)$, and so in this case the two notions coincide. To see however that this is not always the case, consider

$$w = \underset{1}{\mathtt{a}}\,\underset{2}{\mathtt{b}}\,\underset{3}{\mathtt{a}}\,\underset{4}{\mathtt{a}}\,\underset{5}{\mathtt{b}}\,\underset{6}{\mathtt{a}}\,\underset{7}{\mathtt{b}}\,\underset{8}{\mathtt{a}}\,\underset{9}{\mathtt{a}}\,\underset{10}{\mathtt{b}}\,\underset{11}{\mathtt{a}}\,\underset{12}{\mathtt{a}}\,\underset{13}{\mathtt{b}}\,.$$

This string contains exactly three maximum repeats, $[1\,..\,6]$, $[6\,..\,11]$, and $[9\,..\,13]$, but of these only the first two constitute an SNE repeat, `abaaba` $= w[1\,..\,6] = w[6\,..\,11]$ — this is because the repeating substring `abaab` $= w[9\,..\,13]$ is a substring of `abaaba`. Thus in a sense the idea of a maximum repeat is more precise. Note also that the maximum repeat `abaab` occurs only once! This is because its two other occurrences are embedded in maximum repeats `abaaba` and are therefore not separately reported. We have

Proposition 1. *A maximum repeat $[i\,..\,j]$ in w is reported as a component of an SNE repeat if and only if $w[i\,..\,j]$ does not occur as a proper substring in any other maximum repeat.*

We call a letter that occurs only once in a string a singleton. Clearly, all singletons represent minimum unique substrings. In order to avoid this trivial case, we assume from now on that, unless otherwise specified, there are no singletons in the strings we consider. This means that each letter has at least two occurrences and so implies two things. First, any minimum unique substrings has length at least two. Second, each letter is part of a maximum repeat. The latter assertion is due to the fact that any repeat can be extended to a maximum repeat.

4 Duality

Minimum unique substrings and maximum repeats are, in a certain sense, to be made precise, dual concepts. The investigation of this duality is the topic of this section.

Assume $[i\,..\,j]$ is a minimum unique substring of w. That means the substring $w[i\,..\,j-1]$ occurs again in w whereas $w[i\,..\,j]$ does not. We can therefore extend $[i\,..\,j-1]$ to the left, by zero or more positions, in order to obtain a maximum repeat $[k\,..\,j-1]$, with $1 \leq k \leq i$. Symmetrically, there is ℓ, $j \leq \ell \leq n$, such that $[i+1\,..\,\ell]$ is a maximum repeat of w; see Fig. 3.

Notice that the above reasoning works even when the interval $[i\,..\,j]$ is at the beginning or at the end of w. Indeed, if $i = 1$, then $[i\,..\,j-1]$ is a maximum repeat and hence $k = i$. The case when $j = n$ is symmetric.

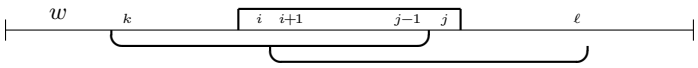

Fig. 3. The minimum unique substring $[i\,..\,j]$ implies the existence of the maximum repeats $[k\,..\,j-1]$ and $[i+1\,..\,\ell]$.

For the example in Fig. 2, $w =$ `abaababa`, consider the minimum unique substring $[5\,..\,7]$. We have that $[4\,..\,6]$ is a maximum repeat, that is, $k = 4$. Also, $[6\,..\,8]$ is a maximum repeat, which makes $\ell = 8$ in this case.

Conversely, assume a maximum repeat $[i\,..\,j]$ of w. Then the substring $w[i\,..\,j]$ occurs again in w whereas $w[i-1\,..\,j]$ does not. Therefore, it will include a minimum unique substring $[i-1\,..\,\ell]$, with $\ell \leq j$. Symmetrically, there exists a minimum unique substring $[k\,..\,j+1]$, for some $k \geq i$; see Fig. 4.

In this case, as opposed to the previous one, if the maximum repeat occurs at the beginning or at the end of the string, then one of the two minimum unique substrings is missing. For instance, if $i = 1$, then the minimum unique substring $[i-1\,..\,\ell]$ does not exist in this case.

In Fig. 2, for the maximum repeat $[4\,..\,6]$ we have the minimum unique strings $[3\,..\,4]$ and $[5\,..\,7]$, which makes $k = 5$ and $\ell = 4$.

We summarize the above findings in Proposition 2.

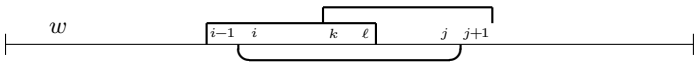

Fig. 4. The maximum repeat $[i\,..\,j]$ implies the existence of the minimum unique substrings $[i-1\,..\,\ell]$ and $[k\,..\,j+1]$.

Proposition 2. *For any string of length n which does not contain singletons we have:*

(i) For any minimum unique substring $[i\,..\,j]$, $1 \le i < j \le n$, there exist k and ℓ with $k \le j-1$, $\ell \ge i+1$ such that $[k\,..\,j-1]$ and $[i+1\,..\,\ell]$ are maximum repeats.

(ii) For any maximum repeat $[i\,..\,j]$, $1 \le i < j \le n$, we have:

(ii.1) If $i \ge 2$, then there exists $\ell \le j$ such that $[i-1\,..\,\ell]$ is a minimum unique substring.

(ii.2) If $j \le n-1$, then there exists $k \ge i$ such that $[k\,..\,j+1]$ is a minimum unique substring.

The relation between minimum unique substrings and maximum repeats is even stronger than the statement of Proposition 2. To start with, it is clear that maximum repeats cannot be included in each other. Therefore, sorting the maximum repeats by their starting positions gives the same ordering as sorting by the ending position. The same property holds for minimum unique substrings. For an example, see Fig. 5.

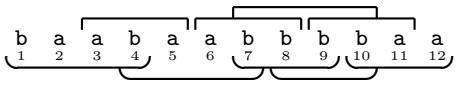

Fig. 5. The maximum repeats (half ovals: $[1\,..\,4]$, $[4\,..\,7]$, $[7\,..\,9]$, $[8\,..\,10]$, $[10\,..\,12]$) and minimum unique substrings (half rectangles: $[3\,..\,5]$, $[6\,..\,8]$, $[7\,..\,10]$, $[9\,..\,11]$) of the string `baabaabbbbaa`.

We have seen that the existence of an object of one type implies the existence of two objects of the other type. (With the exception of maximum repeats at the beginning or end of the string.) The next propositions show that the intervals of the two types are interleaved.

Proposition 3. *For any string of length n which does not contain singletons, an interval $[i\,..\,j]$ is a minimum unique substring if and only if there exist k and ℓ with $1 \le k \le i$, $j \le \ell \le n$, such that*

(i) $[k\,..\,j-1]$ and $[i+1\,..\,\ell]$ are maximum repeats and

(ii) there is no maximum repeat $[k'\,..\,\ell']$ with $k < k' \le i$ or $j \le \ell' < \ell$.

Proof. Assume $[i\,..\,j]$ is a minimum unique substring. The existence of k and ℓ with the property (i) follows from Proposition 2. Assume there exists a maximum repeat $[k'\,..\,\ell']$ with $k < k' \leq i$. As it cannot be included in another maximum repeat, such as $[k\,..\,j-1]$, it must be that $\ell' \geq j$. But then it includes $[i\,..\,j]$ contradicting its uniqueness. The existence of a maximum repeat $[k'\,..\,\ell']$ with $j \leq \ell' < \ell$ causes a similar contradiction.

Conversely, assume (i)-(ii). If $[i\,..\,j]$ is not unique, then it can be extended to a maximum repeat $[k'\,..\,\ell']$, with $k < k' \leq i$ and $j \leq \ell' < \ell$, contradicting (ii). Thus $[i\,..\,j]$ is unique. It is also minimum as both $[i\,..\,j-1]$ and $[i+1\,..\,j]$ are repeats. □

A dual results is stated as Proposition 4 and has a similar proof. The only difference comes from the fact, discussed above, that a maximum repeat at the end of the string implies the existence of only one unique substring.

Proposition 4. *For any string of length n which does not contain singletons, an interval $[i\,..\,j]$ is a maximum repeat if and only if there exist $k \geq i$ (assuming $i \geq 2$) and $\ell \leq j$ (assuming $j \leq n-1$) such that*

(i) $[k\,..\,j+1]$ and $[i-1\,..\,\ell]$ are minimum unique substrings and
(ii) there is no minimum unique substring $[k'\,..\,\ell']$ with $i \leq k' < k$ or $\ell < \ell' \leq j$.

Proposition 3 proves that every minimum unique substring starts in-between two consecutive left ends of maximum repeats and ends between their right ends. Proposition 4 proves that the converse is also true, with the exception of the first and last maximum repeats. Therefore, the sorted lists of maximum repeats and minimum unique substrings are interleaved, with maximum repeats both starting and ending the sequence. Theorem 1 gives a precise picture of the dual relation (for an example see Fig. 5).

Theorem 1. *For any string of length n which does not contain singletons, if the maximum repeats, sorted increasingly by their left ends (or right ends, the same thing) are $[i_\ell\,..\,j_\ell], 1 \leq \ell \leq k+1$, then the minimum unique substrings (also sorted increasingly, by either end) are $[i_{\ell+1}-1\,..\,j_\ell+1], 1 \leq \ell \leq k$.*

Several remarks are in order. First, it is clear from the statement of Theorem 1 that we can also obtain the maximum repeats from the minimum unique intervals. Second, singletons represent trivial minimum unique substrings and no maximum repeat can span a singleton. Therefore, singletons divide the string into substrings not containing any singletons and so all our results extend immediately to the general that includes singletons. Third, the number of maximum repeats is always one larger than the number of minimum unique substrings.

5 Algorithms

We focus in this section on algorithms to compute minimum unique substrings. It should be noted that a direct consequence of Theorem 1 is that any algorithm for

MinUniqueFromMaxRepeat(w, maximum repeats)

1. sort maximum repeats in ascending order: $[i_1 \,..\, j_1], \ldots, [i_{k+1} \,..\, j_{k+1}]$
2. **for** ℓ **from** 1 **to** k **do**
3. **output** $[i_{\ell+1} - 1 \,..\, j_\ell + 1]$
4. **return**

Fig. 6. Finding all minimum unique substrings using the maximum repeats.

MaxRepeatFromMinUnique(w, minimum unique substrings)

1. sort minimum unique substrings in ascending order: $[i_1 \,..\, j_1], \ldots, [i_k \,..\, j_k]$
2. **output** $[1 \,..\, j_1 - 1]$
3. **for** ℓ **from** 1 **to** $k - 1$ **do**
4. **output** $[i_\ell + 1 \,..\, j_{\ell+1} - 1]$
5. **output** $[i_k + 1 \,..\, n]$
6. **return**

Fig. 7. Finding all maximum repeats using the minimum unique substrings.

computing minimum unique substrings is easily transformed into an algorithm for computing maximum repeats and vice versa; the corresponding procedures are shown in Figs. 6 and 7.

We give next several algorithms to compute both maximum repeats and minimum unique substrings directly. The first two are, in some sense, straightforward. They make use of the *inverse suffix array*, ISA, defined by $\mathsf{ISA}[j] = i$ iff $\mathsf{SA}[i] = j$. The inverse suffix array helps in finding the position of a given suffix in the suffix array in constant time. We shall denote the arrays corresponding to the reverse string, w_r, by SA_r, ISA_r, and LCP_r. These algorithms use two indexes, for both w and w_r. In order to simplify the code, we shall assume that $\mathsf{LCP}[1] = \mathsf{LCP}[n+1] = 0$.

The algorithm TwoIndex-MaxRepeat in Fig. 8 starts by computing five arrays, which can be done in linear time and space as previously mentioned. Then, at the end of step 5, $w[i \,..\, j]$ is the longest prefix of $\mathsf{suf}[i]$ that is a repeat; it has length *lcp*. To see whether $[i \,..\, j]$ is a maximum repeat, the corresponding check in w_r is performed; ISA_r is needed to find the place where the corresponding suffix of w_r is in SA_r. The length of its longest prefix that is a repeat is lcp_r. If they are the same (step 7), then $[i \,..\, j]$ is a maximum repeat.

The algorithm TwoIndex-MinUnique in Fig. 9 is similar. After detecting that $[i \,..\, j]$ is a maximum repeat in step 7, it finds the minimum unique substring ending at $j + 1$ by computing its length in the reverse index.

The above algorithms are straightforward as they use the two indexes to compute the correct length of the strings searched for. However, the computation of the data structures involved is by far the most time consuming step. It is a great disadvantage that so many arrays are needed. We give next two algorithms which not only require a single index but also are very simple.

The MaxRepeat algorithm, shown in Fig. 10, uses the property that any maximum repeat $[i \,..\, j]$ is the longest among all repeated prefixes ending at j of

```
TwoIndex-MaxRepeat(w)
1.  compute SA, LCP, SA_r, ISA_r, LCP_r
2.  for k from 1 to n do
3.      lcp = max(LCP[k], LCP[k + 1])
4.      if (lcp > 0) then
5.          i ← SA[k]; j ← i + lcp − 1; j_r ← n − j + 1
6.          lcp_r ← max(LCP_r[ISA_r[j_r]], LCP_r[ISA_r[j_r] + 1])
7.          if (lcp = lcp_r) then
8.              output [i . . j]
9.  return
```

Fig. 8. Finding all maximum repeats by indexing both w and w_r.

```
TwoIndex-MinUnique(w)
1.  compute SA, LCP, SA_r, ISA_r, LCP_r
2.  for k from 1 to n do
3.      lcp = max(LCP[k], LCP[k + 1])
4.      if (lcp > 0) then
5.          i ← SA[k]; j ← i + lcp − 1; j_r ← n − j + 1
6.          lcp_r ← max(LCP_r[ISA_r[j_r]], LCP_r[ISA_r[j_r] + 1])
7.          if (lcp = lcp_r) and (j < n) then
8.              ℓ ← max(LCP_r[ISA_r[j_r − 1]], LCP_r[ISA_r[j_r − 1] + 1])
9.              output [j − ℓ + 2 . . j + 1]
10. return
```

Fig. 9. Finding all minimum unique substrings by indexing both w and w_r.

all suffixes of w. The maximum repeat $[i\,..\,j]$ will be stored in the end by setting $\textsc{maxRep}[j] = i$. Therefore, we initialize the maxRep array with values larger than any valid ones in step 3. In step 5 we compute the longest repeated prefix of each suffix and update maxRep in step 6 if a longer one is found. We remark that this algorithm is conceptually much simpler than the two algorithms PSY2 proposed for SNE repeats in [22], and that moreover, unlike the PSY2 algorithms, its execution time has no explicit dependence on alphabet size.

```
MaxRepeat(w)
1.  compute SA, LCP
2.  for i from 1 to n do
3.      maxRep[i] ← n + 1
4.  for i from 1 to n do
5.      lcp ← max(LCP[i], LCP[i + 1])
6.      maxRep[SA[i] + lcp − 1] ← min(maxRep[SA[i] + lcp − 1], SA[i])
7.  return maxRep
```

Fig. 10. A simple algorithm for finding all maximum repeats.

The MinUnique algorithm, shown in Fig. 11, is based on a similar observation: Any minimum unique substring $[i\,..\,j]$ is the shortest among all unique prefixes ending at j of all suffixes of w. The minimum unique substring $[i\,..\,j]$ will be stored in the end by setting $\textsc{minUnique}[j] = i$. Therefore, we initialize the minUnique array with values smaller than any valid ones in step 3. In step 5 we compute the longest repeated prefix of each suffix. The shortest unique prefix is one position larger, except when the longest repeated prefix reaches the end of w. (To avoid checking this at every step, we allow minUnique to have $n + 1$ components; its last one will be ignored.) We update minUnique in step 6 if a shorter one is found.

Since the initialization in steps 2-3 is often done automatically when the array is created, the code of MinUnique has, aside from computing the index, only three lines.

```
MinUnique(w)
1.  compute SA, LCP
2.  for i from 1 to n do
3.      minUnique[i] ← 0
4.  for i from 1 to n do
5.      lcp ← max(LCP[i], LCP[i + 1])
6.      minUnique[SA[i] + lcp] ← max(minUnique[SA[i] + lcp], SA[i])
7.  return minUnique
```

Fig. 11. A simple algorithm for finding all minimum unique substrings.

Alternatively, we could store the minimum unique substring by their left end, that is, $[i\,..\,j]$ is stored by setting $\text{minUnique2}[i] = j$. This is done in the MinUnique-LeftEnd algorithm, shown in Fig. 12. We compute the shortest unique substrings with a given left end (steps 2-3) and then identify and retain the minimum ones (steps 4-8). We assume $\text{minUnique2}[n+1] = 1$.

```
MinUnique-LeftEnd(w)
1.  compute SA, LCP
2.  for i from 1 to n do
3.      minUnique2[SA[i]] ← 1 + max(LCP[i], LCP[i + 1])
4.  i ← 1
5.  while (i ≤ n) and (i + minUnique2[i] ≤ n + 1) do
6.      while (minUnique2[i] > minUnique2[i + 1]) do
7.          minUnique2[i] ← 0
8.          i ← i + 1
9.  return minUnique2
```

Fig. 12. Finding minimum unique substrings by their left end.

A similar idea works also for computing maximum repeats.

Clearly, all the above algorithms, for computing minimum unique substrings or maximum repeats, run in linear time, independent of the size of the alphabet.

Theorem 2. *Minimum unique substrings can be computed in linear time, independent of the size of the input alphabet.*

6 De Bruijn strings

Strings having the highest possible number of minimum unique substrings are expected to have a high complexity and we investigate them in this section. In view of Theorem 1, this is the same as considering strings with the highest number of maximum repeats. The notion of the highest number of substrings brings immediately in mind the de Bruijn strings [6] but a bit of work is required to establish the precise connection.

Considering an alphabet of k letters and a substring length ℓ, de Bruijn strings have as substrings all possible distinct strings of length ℓ and have minimum possible length for this property, that is, $k^\ell + \ell - 1$. Two examples are shown in Figs. 13 and 14, for $k = 2, \ell = 4$ and $k = 3, \ell = 2$, respectively. Their maximum repeats and minimum unique substrings are also shown.

We show that de Bruijn strings, and only those, satisfy our properties.

Theorem 3. *A string of length $k^\ell + \ell - 1$ over an alphabet of size k has a maximum number of minimum unique substrings (or, equivalently, maximum repeats) if and only if it is a de Bruijn string. In such a case, its number of minimum unique substrings is k^ℓ.*

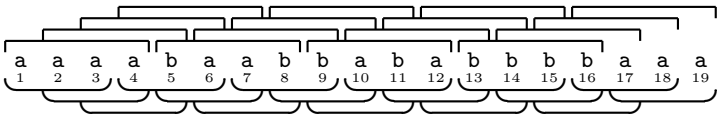

Fig. 13. The maximum repeats and minimum unique substrings of the binary de Bruijn string `aaaabaabbababbbbaaa`.

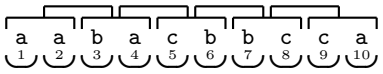

Fig. 14. The maximum repeats and minimum unique substrings of the ternary de Bruijn string `aabacbbcca`.

Proof. A de Bruijn string has k^ℓ minimum unique substrings by definition. Assume there is a k-letter string w of the same length that has more than k^ℓ. The fact that minimum substrings cannot be substrings of each other implies two things. First, w must have some minimum unique substrings of length higher than ℓ; denote the maximum such length by $\ell' > \ell$. Second, w cannot have more than $|w| - \ell' + 1$ minimum unique substrings. Since $|w| - \ell' + 1 = k^\ell + \ell - \ell' < k^\ell$, a contradiction is obtained. □

7 Conclusions and future research

We have clarified the relation between minimum unique substrings and maximum repeats and our algorithms for finding both types are very simple, making their computation almost a byproduct of the suffix array construction. It remains an open problem to find algorithms that do not employ text indexes.

Minimum unique substrings are used by [10] to design DNA oligonucleotides. Precisely, a heuristic algorithm is given to predict the location of oligonucleotides, based on counting right ends of minimum unique prefixes. Rigorously, what is needed is finding approximate unique substrings, that is, substrings which are far from all the other ones with respect to a given distance. Considering the Hamming distance, d, and a fixed $k \geq 0$, a substring x of w is called *k-unique* if $d(x, y) > k$, for any substring $y \neq x$ of w. A superstring of a k-unique string is also k-unique. Therefore, *minimum k-unique substrings* are well defined. The (exact) case we investigated corresponds to $k = 0$. Finding practical algorithms (that is, which can handle large genomic sequences) for computing minimum k-unique substrings is an interesting research direction.

References

1. M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch, **Replacing suffix trees with enhanced suffix arrays**, *J. Discrete Algorithms 2* (2004) 53–86.

2. A. Apostolico and F. P. Preparata, **Optimal off-line detection of repetitions in a string**, *Theoret. Comput. Sci. 22* (1983) 297–315.
3. A. Bakalis, C. S. Iliopoulos, C. Makris, S. Sioutas, E. Theodoridis, A. K. Tsakalidis and K. Tsichlas, **Locating maximum multirepeats in multiple strings under various constraints**, *The Computer Journal 50–2* (2007) 178–185.
4. G. S. Brodal, R. B. Lyngsø, C. N. S. Pedersen and J. Stoye, **Finding maximum pairs with bounded gap**, *J. Discrete Algorithms 1* (2000) 77–103.
5. J. Craig Venter et al., **The Sequence of the Human Genome**, *Science 291 (5507)* (2001) 1304–1351.
6. N. G. de Bruijn, **A combinatorial problem**, *Proc. Konin. Neder. Akad. Wet. 49* (1946) 758–764.
7. M. Crochemore, **An optimal algorithm for computing all the repetitions in a word**, *Inform. Process. Lett. 12–5* (1981) 244–248.
8. M. Crochemore, C. S. Iliopoulos, M. Mohamed and M.-F. Sagot, **Longest repeats with a block of k don't cares**, *Theoret. Comput. Sci. 362–1* (2006) 248–254.
9. F. Franek, W. F. Smyth and Y. Tang, **Computing all repeats using suffix arrays**, *J. Automata, Languages and Combinatorics 8–4* (2003) 579–591.
10. S. Gräf, F. G. G. Nielsen, S. Kurtz, M. A. Huynen, E. Birney, H. Stunnenberg and P. Flicek, **Optimized design and assessment of whole genome tiling arrays**, *Bioinformatics 23–13* (2007) i195–i204.
11. D. Gusfield, *Algorithms on Strings, Trees, and Sequences*, Cambridge University Press (1997) 534 pp.
12. B. Haubold, N. Pierstorff, F. Möller and T. Wiehe, **Genome comparison without alignment using shortest unique substrings**, *BMC Bioinformatics 6* (2005) 123–133.
13. C. S. Iliopoulos, W. F. Smyth and M. Yusufu, **Faster algorithms for computing maximum multirepeats in multiple sequences**, *Fundamenta Informaticae 97–3* (2009) 311-320.
14. T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park, **Linear-time longest-common-prefix computation in suffix arrays and its applications**, *Proc. 12th Annual Symp. Combinatorial Pattern Matching*, LNCS 2089 (2001) 181–192.
15. E. S. Lander et al., **Initial sequencing and analysis of the human genome**, *Nature 409* (2001) 860–921.
16. A. Lefebvre and T. Lecroq, **A heuristic for computing repeats with a factor oracle: application to biological sequences**, *Internat. J. Comput. Math. 79–12* (2002) 1303–1315.
17. M. G. Main and R. J. Lorentz, **An $O(n \log n)$ algorithm for finding all repetitions in a string**, *J. Algorithms 5* (1984) 422–432.
18. U. Manber and G. W. Myers, **Suffix arrays: a new method for on-line string searches**, *Proc. First Annual ACM-SIAM Symp. Discrete Algs.* (1990) 319–327.
19. K. Narisawa, S. Inenaga, H. Bannai and M. Takeda, **Efficient computation of substring equivalence classes with suffix arrays**, *Proc. 18th Annual Symp. Combinatorial Pattern Matching*, LNCS 4580 (2007) 340–351.
20. G. Nong, S. Zhang and W. H. Chan, **Linear time suffix array construction using D-critical substrings**, *Proc. 20th Annual Symp. Combinatorial Pattern Matching*, LNCS 5577 (2009) 54–67.
21. S. J. Puglisi, W. F. Smyth and A. H. Turpin, **A taxonomy of suffix array construction algorithms**, *ACM Computing Surveys 39–2* (2007) 1–31.
22. S. J. Puglisi, W. F. Smyth and M. Yusufu, **Fast, practical algorithms for computing all the repeats in a string**, *Math. Comput. Sci. 3* (2010) 373–389.

23. S. J. Puglisi and A. H. Turpin, **Space-time tradeoffs for longest-common-prefix array computation**, *Proc. 19th Internat. Symp. Algs. and Computation*, LNCS 5369 (2008) 124–135.
24. S. Rahmann, **Fast large scale oligonucleotide selection using the longest common factor approach**, *J. Bioinformatics and Computational Biology 1–2* (2003) 343–361.
25. B. Smyth, *Computing Patterns in Strings*, Pearson Addison-Wesley (2003) 423 pp.
26. A. Thue, **Über unendliche zeichenreihen**, *Norske Vid. Selsk. Skr. I. Mat. Nat. Kl. Christiana 7* (1906) 1–22.
27. P. Weiner, **Linear pattern matching algorithms**, *Proc. 14th Annual IEEE Symp. Switching and Automata Theory* (1973) 1–11.
28. J. Zheng, T. J. Close, T. Jiang and S. Lonardi, **Efficient selection of unique and popular oligos for large EST databases** *Proc. 14th Annual Symp. Combinatorial Pattern Matching*, LNCS 2676 (2003) 273–283.