Cover array string reconstruction

Maxime Crochemore, Costas S. Iliopoulos, Solon P. Pissis, German Tischler

Cover array string reconstruction

Maxime Crochemore[1,2], Costas S. Iliopoulos[1,3],
Solon P. Pissis[1], German Tischler[1,4]

[1] Dept. of Computer Science, King's College London, London WC2R 2LS, UK
{mac,csi,pississo,tischler}@dcs.kcl.ac.uk
[2] Université Paris-Est, France
[3] Digital Ecosystems & Business Intelligence Institute, Curtin University
GPO Box U1987 Perth WA 6845, Australia
[4] Newton Fellow

Abstract. A proper factor u of a string y is a cover of y if every letter of y is within some occurrence of u in y. The concept generalises the notion of periods of a string. An integer array C is the minimal-cover (resp. maximal-cover) array of y if $C[i]$ is the minimal (resp. maximal) length of covers of $y[0\,..\,i]$, or zero if no cover exists.
In this paper, we present a constructive algorithm checking the validity of an array as a minimal-cover or maximal-cover array of some string. When the array is valid, the algorithm produces a string over an unbounded alphabet whose cover array is the input array. All algorithms run in linear time due to an interesting combinatorial property of cover arrays: the sum of important values in a cover array is bounded by twice the length of the string.

Introduction

The notion of periodicity in strings is well studied in many fields like combinatorics on words, pattern matching, data compression and automata theory (see [11, 12]), because it is of paramount importance in several applications, not to talk about its theoretical aspects.

The concept of quasiperiodicity is a generalisation of the notion of periodicity, and was defined by Apostolico and Ehrenfeucht in [2]. In a periodic repetition the occurrences of the single periods do not overlap. In contrast, the quasiperiods of a quasiperiodic string may overlap. We call a proper factor u of a nonempty string y a cover of y, if every letter of y is within some occurrence of u in y. In this paper, we consider the so-called *aligned covers*, where the cover u of y needs to be a border (i.e. a prefix and a suffix) of y. The array C is called the *minimal-cover* (resp. *maximal-cover*) *array* of the string y of length n, if for each i, $0 \leq i < n$, $C[i]$ stores either the length of the shortest (resp. longest) cover of $y[0\,..\,i]$, when such

a cover exists, or zero otherwise. In particular, we do not consider a string to be a cover of itself.

Apostolico and Breslauer [1, 4] gave an online linear runtime algorithm computing the minimal-cover array of a string. In their definition, a string is a cover of itself, but it is straightforward to modify their algorithm to accommodate our definition. Li and Smyth [10] provided an online linear runtime algorithm for computing the maximal-cover array.

In this paper, we present a constructive algorithm checking if an integer array is the minimal-cover or maximal-cover array of some string. When the array is valid, the algorithm produces a string over an unbounded alphabet whose cover array is the input array. For our validity checking algorithm, we use the aforementioned algorithms that compute cover arrays.

All algorithms run in linear time. This is essentially due to a combinatorial property of cover arrays: the sum of important values in a cover array is bounded by twice the length of the string.

The result of the paper completes the series of algorithmic characterisations of data structures that store fundamental features of strings. They concern Border arrays [6, 7], Parameterized Border arrays [9] and Prefix arrays [5] that stores periods of all the prefixes of a string, as well as the element of Suffix arrays [3, 8] that memorises the list of positions of lexicographically sorted suffixes of the string. The question is not applicable to complete Suffix trees or Suffix automata since the relevant string is part of these data structures. The algorithms may be regarded as reverse engineering processes and, beyond their obvious theoretical interest, they are useful to test the validity of some constructions. Their linear runtime is an important part of their quality.

The rest of the paper is structured as follows. Section 1 presents the basic definitions used throughout the paper and the problem. In Section 2, we prove some properties of minimal-cover arrays used later for the design or the analysis of algorithms. In Section 3, we describe our constructive cover array validity checking algorithms. Section 4 provides some combinatorially interesting numerical data on minimal-cover arrays.

1 Definitions and Problems

Throughout this paper we consider a string y of length $|y| = n$ on an unbounded alphabet. It is represented as $y[0\,..\,n-1]$. A string w is a *factor* of y if $y = uwv$ for two strings u and v. It is a *prefix* of y if u is empty and a *suffix* of y if v is empty. A string u is a *period* of y if

y is a prefix of u^k for some positive integer k, or equivalently if y is a prefix of uy. The period of y is the shortest period of y. A string x of length m is a *cover* of y if both $m < n$ and there exists a set of positions $P \subseteq \{0, \ldots, n-m\}$ satisfying $y[i \,..\, i+m-1] = x$ for all $i \in P$ and $\bigcup_{i \in P} \{i, \ldots, i+m-1\} = \{0, \ldots, n-1\}$. Note that this requires x to be a prefix as well as a suffix of y. The *minimal-cover array* C of y is the array of integers $C[0 \,..\, n-1]$ for which $C[i]$, $0 \leq i < n$, stores the length of the shortest cover of the prefix $y[0 \,..\, i]$, if such a cover exists, or zero otherwise. The *maximal-cover array* C^{M} stores longest cover at each position instead. The following table provides the minimal-cover array C and the maximal-cover array C^{M} of the string $y = \texttt{abaababaabababaababaaba}$.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
$y[i]$	a	b	a	a	b	a	b	a	a	b	a	b	a	a	b	a	a	b	a	b	a	a	b	a
$C[i]$	0	0	0	0	0	3	0	3	0	5	3	7	3	9	5	3	0	5	3	0	3	9	5	3
$C^{\mathrm{M}}[i]$	0	0	0	0	0	3	0	3	0	5	6	7	8	9	10	11	0	5	6	0	8	9	10	11

We consider the following problems for an integer array A:

Problem 1 (Minimal Validity Problem). Decide if A is the minimal-cover array of some string.

Problem 2 (Maximal Validity Problem). Decide if A is the maximal-cover array of some string.

Problem 3 (Minimal Construction Problem). When A is a valid minimal-cover array, exhibit a string over an unbounded alphabet whose minimal-cover array is A.

Problem 4 (Maximal Construction Problem). When A is a valid maximal-cover array, exhibit a string over an unbounded alphabet whose maximal-cover array is A.

2 Properties of the Minimal-Cover Array

In this section, we assume that C is the minimal-cover array of y. Its first element is 0, as we do not consider a string to be a cover of itself. Next elements are 1 only for prefixes of the form a^k for some letter a. We will use the following fact in our argumentation.

Fact 1 (Transitivity) *If u and v cover y and $|u| < |v|$, then u covers v.*

For the rest of the section we assume that $n > 1$ and prove several less obvious properties of the minimal-cover array.

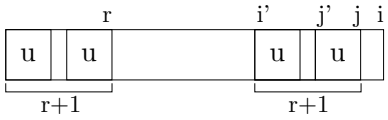

Fig. 1. Case $j - C[j] > i - C[i]$ of Lemma 2 ($i' = i - C[i] + 1$, $j' = j - C[j] + 1$).

Lemma 1. *If* $0 \leq i < n$ *and* $C[i] \neq 0$*, then* $C[C[i] - 1] = 0$.

Proof. Immediate from Fact 1. □

Lemma 2. *Let* i *and* j *be positions such that* $j < i$*,* $j - C[j] \geq i - C[i]$*,* $C[i] \neq 0$ *and* $C[j] \neq 0$*. Furthermore let* $r = j-(i-C[i]+1)$*. If* $i-C[i] = j-C[j]$ *then* $C[r] = 0$*, otherwise if* $i-C[i] < j-C[j]$*, then* $C[r] = C[j]$.

Proof. First assume that $i-C[i] = j-C[j]$. Then $C[j-(i-C[i]+1)] = C[j-(j-C[j]+1)] = C[C[j]-1] = 0$ according to Lemma 1. Now assume that $i - C[i] < j - C[j]$. This situation is depicted in Figure 1. The string $u = y[j-C[j]+1\,..\,j]$ of length $C[j]$ covers the string $y[0\,..\,j]$. By precondition ($j < i$, $j - C[j] > i - C[i]$) u also covers $y[0\,..\,r]$, as there exists an occurrence of u at position $r - C[j] + 1$. Thus we have $C[r] \leq C[j]$. The assumption $C[r] < C[j]$ leads to a contradiction to the minimality of C. Thus we have $C[r] = C[j]$. □

Lemma 3. *Let* i *and* j *be positions such that* $j < i$ *and* $j-C[j] < i-C[i]$*. Then* $(i - C[i]) - (j - C[j]) > C[j]/2$.

Proof. For ease of notation let $p = C[i]$, $q = C[j]$ and $r = (i-p)-(j-q)$. Assume the statement does not hold, i.e. $r \leq \frac{q}{2}$. Let $u = y[0\,..\,r-1]$. Then due to the overlap of $y[i-p+1\,..\,i]$ and $y[j-q+1\,..\,j]$ both $y[0\,..\,p-1]$ and $y[0\,..\,q-1]$ are powers of u. Let $y[0\,..\,q-1] = u^e$ for some exponent e. Observe that $e = q/r \geq q/(q/2) = 2$. However $y[0\,..\,q-1]$ is also covered by $v = u^{1+e-\lfloor e \rfloor}$. As $|v| < q$ we obtain a contradiction. □

Definition 1. *A position* $j \neq 0$ *of* C *is called* totally covered*, if there is a position* $i > j$ *for which* $C[i] \neq 0$ *and* $i - C[i] + 1 \leq j - C[j] + 1 < j$.

Let C^{p} be obtained from C by setting $C[i] = 0$ for all totally covered indices i on C. We call C^{p} the *pruned minimal-cover array* of y. The next table shows the pruned minimal-cover array of the example string above.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
$y[i]$	a	b	a	a	b	a	b	a	a	b	a	b	a	a	b	a	a	b	a	b	a	a	b	a
$C[i]$	0	0	0	0	0	3	0	3	0	5	3	7	3	9	5	3	0	5	3	0	3	9	5	3
$C^{\mathrm{p}}[i]$	0	0	0	0	0	3	0	0	0	0	0	0	0	9	5	0	0	0	0	0	0	9	5	3

Lemma 4. *The sum of the elements of C^{p} does not exceed $2n$.*

Proof. Let $I_i = \{i - C[i] + 1, i - C[i] + 2, \dots i\}$ for $i = 0, \dots, n-1$ if $C[i] \neq 0$ and $I_i = \emptyset$ otherwise. Let I'_i denote the lower half of I_i (if $C[i]$ is uneven, the middle element is included). According to Lemma 3, $i \neq j$ implies $I'_i \cap I'_j = \emptyset$. Thus the relation $\sum_{i=0}^{n-1} |I'_i| \leq n$ holds, which in turn implies $\sum_{i=0}^{n-1} |I_i| \leq \sum_{i=0}^{n-1} 2|I'_i| \leq 2n$. □

The bound of Lemma 4 is asymptotically tight. For an integer $k > 1$, let $x_k = (a^k b a^{k+1} b)^{n/(2k+3)}$. For $k = 2$ and $n = 23$ we get:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
$y[i]$	a	a	b	a	a	a	b	a	a	b	a	a	a	b	a	a	b	a	a	a	b	a	a
$C^{\mathrm{p}}[i]$	0	0	0	0	0	0	0	0	5	0	0	0	0	7	0	5	0	0	0	0	7	0	5

It is straightforward to see that all segments of length $2k + 3$ of C^{p} contain the values $2k + 1$ and $2k + 3$, except at the beginning of the string. Therefore the sum of elements in C^{p} is $(4k + 4)(\frac{n}{2k+3} - 1)$, which tends to $2n$ when k (and n) goes to infinity.

3 Reverse Engineering a Cover Array

We solve the stated problems in three steps: array transformation, string inference and validity checking.

Transforming Maximal to Minimal-Cover Arrays We first show how to transform a maximal-cover array into a minimal-cover array in linear time. The following algorithm MAXTOMIN converts the maximal-cover array C of y to its minimal-cover array in linear time.

MAXTOMIN(C, n)

```
1  for i ← 0 to n − 1 do
2      if C[i] ≠ 0 and C[C[i] − 1] ≠ 0 then
3          C[i] ← C[C[i] − 1]
```

The algorithm works in the following way. Assume the for loop is executing for some $i > 0$. At this time the prefix $C[0\,..\,i-1]$ of the array has been converted to a minimal-cover array and we want to determine the value

of $C[i]$. If $C[i]$ is zero then there is no cover and we leave the value as it is. If $C[i]$ is not zero, then we know from Fact 1 that if there is a shorter cover, then it covers $y[0\,..\,C[i]-1]$. Thus we look up $C[C[i]-1]$ to see if there is a shorter cover we can substitute for $C[i]$. As the segment before index i is already the minimal-cover array up to this point, we know that such a value is minimal.

String Inference In this step, we assume that the integer array C of length n is the minimal- or maximal-cover array of at least one string. From what we wrote above, we can assume without loss of generality that C is a minimal-cover array.

The nonzero values in C induce an equivalence relation on the positions of every string that has the minimal-cover array C. More precisely, if we find the value $\ell \neq 0$ in position i of C, then this imposes the constraints

$$y[k] = y[i - \ell + 1 + k]$$

for $k = 0, \ldots, \ell - 1$. We say that the positions k and $i - \ell + 1 + k$ are bidirectionally linked. Let the undirected graph $G(V, E)$ be defined by $V = \{0, \ldots, n-1\}$ and

$$E = \bigcup_{i=0,\ldots,n-1} \bigcup_{j=0,\ldots C[i]-1} (\{(j, i - C[i] + 1 + j)\})\,.$$

Then the nonzero values in C state that the letters at positions i and j of any word y such that C is the minimal-cover array of y need to be equal, if i and j are connected in G. According to Lemma 2, we do not lose connectivity between vertices of G, if we remove totally covered indices from C, i.e. the graph induced by C has the same connected components as the one induced by its pruned version C^{p}. The number of edges in the graph induced by C^{p} is bounded by $2n$ according to Lemma 4.

The pruned minimal-cover array C^{p} can be obtained from the minimal-cover array C using the following algorithm Prune in linear time.

```
Prune(C, n)
1  ℓ ← 0
2  for i ← n − 1 downto 0 do
3       if ℓ ≥ C[i] then
4            C[i] ← 0
5       ℓ ← max(0, max(ℓ, C[i]) − 1)
6  return C
```

A non-zero value at index i in C defines an interval $[i - C[i] + 1, i]$. The

algorithm scans C from large to small indices, where the value ℓ stores the minimal lower bound of all the intervals encountered so far. If an interval starting at a smaller upper bound has a greater lower bound than ℓ, we erase the corresponding value in C by setting it to zero. Thus we remove all totally covered indices from C and obtain the pruned array C^{p}.

So far we know how to extract information from the non-zero values of C by computing connected components in a graph which has no more than $2n$ edges. The vertices in each connected component designate positions in any produced string which need to have equal letters. By assigning a different letter to each of these components, we make sure not to violate any constraints set by the zero values in C. The following algorithm MinArrayToString produces a string y from an array A assumed to be a pruned minimal-cover array.

```
MinArrayToString(A, n)
 1  ▷ Produce edges
 2  for i ← 0 to n − 1 do
 3       E[i] ← empty list
 4  for i ← 0 to n − 1 do
 5       for j ← 0 to A[i] − 1 do
 6            E[i − A[i] + 1 + j].add(j), E[j].add(i − A[i] + 1 + j)
 7  ▷ Compute connected components by Depth First Search
 8  ▷ and assign letters to output string
 9  (S, ℓ) ← (empty stack, −1)
10  for i ← 0 to n − 1 do
11       if y[i] is undefined then
12            S.push(i)
13            ℓ ← ℓ + 1
14            while not S.empty() do
15                 p ← S.pop()
16                 y[p] ← ℓ
17                 for each element j of E[p] do
18                      if y[j] is undefined then
19                           S.push(j)
20  return y
```

The first two for loops produce the edges E in the graph G induced by A, where we implement the transition relation by assigning a linear list of outgoing edges to each vertex. The third for loop computes the connected components in the graph by using depth first search and assigns the letters to the output string. Each connected component is assigned a

different letter. The runtime of the algorithm is linear in the number of edges of the graph, which is bounded by $2n$.

Theorem 1. *The problems* Minimal Construction Problem *and* Maximal Construction Problem *are solved in linear time by the algorithm* MinArrayToString *and the sequence of algorithms* MaxToMin *and* MinArrayToString *respectively.*

Validity Checking In the third step we use the MinArrayToString algorithm as a building block for our validity checking algorithm. Thus we have to ensure some basic constraints so that the algorithm does not firstly access any undefined positions in the input and secondly runs in linear time. As a first step, we have to make sure that the algorithm will not try to define edges for which at least one vertex number is not valid. This check is performed by the following algorithm Precheck, which runs in linear time.

Precheck(A, n)

```
1  for i ← 0 to n − 1 do
2      if i − A[i] + 1 < 0 then
3          return false
4  return true
```

If Precheck returns true, then MinArrayToString will only generate edges from valid to valid vertices. If we are checking for validity of a maximal-cover array, we then pass the array through the algorithm MaxToMin as a next step. In both cases (minimal and maximal) the next step is to prune the array using the algorithm Prune. After this, we can call the algorithm MinArrayToString with the so-modified array, but it may not run in linear time, as the constraints imposed by Lemma 3 may not hold if the original input array is invalid. We avoid this situation with the following algorithm Postcheck.

Postcheck(A, n)

```
1  j ← −1
2  for i ← 0 to n − 1 do
3      if A[i] ≠ 0 then
4          if j ≠ −1 and (i − A[i]) − (j − A[j]) ≤ ⌊A[j]/2⌋ then
5              return false
6          j ← i
7  return true
```

If Postcheck returns false, then the input array was invalid. Otherwise

we can call MinArrayToString and be sure that it will run in linear time. At this point we have obtained a string from the input array in linear time. We know that if the input array is valid, the minimal- or maximal- (depending on what kind of input we are given) cover array of this string matches the input. If the input array is not valid, we cannot obtain it by computing the minimal- or maximal-cover array from the obtained string. Thus we can check whether an array A is a valid maximal-cover array using the following algorithm CheckMaximal.

CheckMaximal(A, n)
1 **if** Precheck$(A, n) =$ false **then**
2 **return** false
3 $A \leftarrow$ MaxToMin(A, n)
4 $A \leftarrow$ Prune(A, n)
5 **if** Postcheck$(A, n) =$ false **then**
6 **return** false
7 $y \leftarrow$ MinArrayToString(A, n)
8 **if** the maximal-cover array of y equals A **then**
9 **return** true
10 **else return** false

The algorithm CheckMinimal for checking whether an array A is a valid minimal-cover array is obtained from CheckMaximal by removing the call to the function MaxToMin in line 3 and checking whether the minimal instead of the maximal-cover array of the string y equals A in line 8.

Theorem 2. *The problems* Minimal Validity Problem *and* Maximal Validity Problem *are solved by the algorithms* CheckMinimal *and* CheckMaximal *respectively in linear time.*

4 Experiments and Numerical Results

Figure 2 shows the maximal ratio of sums of elements of pruned minimal-cover array, for all words over a two-letter alphabet, using even word lengths 8 to 30. These ratios are known to be smaller than 2 by Lemma 4. However, values close to this bound are not observed for small word length.

We were able to verify the linear runtime of our algorithm in experiments. The implementation for the CheckMinimal function is available at the Website http://www.dcs.kcl.ac.uk/staff/tischler/src/recovering-0.0.0.tar.bz2, which is set up for maintaining the source code and the documentation.

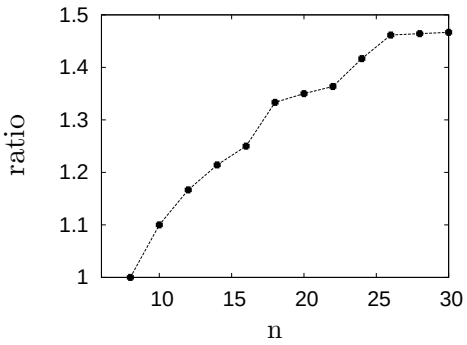

Fig. 2. Maximal ratio of sum over pruned minimal-cover array and word length for words over the binary alphabet of even length 8 to 30

5 Conclusion

In this paper, we have provided linear runtime algorithms for checking the validity of minimal- and maximal-cover arrays and algorithms to infer strings from valid minimal- and maximal-cover arrays. The linear time inference of strings using the least possible alphabet size from cover arrays remains an open problem.

References

1. A. Apostolico and D. Breslauer. Of periods, quasiperiods, repetitions and covers. In J. Mycielski, G. Rozenberg, and A. Salomaa, editors, *Structures in Logic and Computer Science*, volume 1261 of *Lecture Notes in Computer Science*, pages 236–248. Springer, 1997.
2. A. Apostolico and A. Ehrenfeucht. Efficient detection of quasiperiodicities in strings. *Theoretical Computer Science*, 119(2):247–265, 1993.
3. H. Bannai, S. Inenaga, A. Shinohara, and M. Takeda. Inferring strings from graphs and arrays. In B. Rovan and P. Vojtás, editors, *Mathematical Foundations of Computer Science*, volume 2747 of *Lecture Notes in Computer Science*, pages 208–217. Springer, 2003.
4. D. Breslauer. An on-line string superprimitivity test. *Information Processing Letters*, 44(6):345–347, 1992.
5. J. Clement, M. Crochemore, and G. Rindone. Reverse engineering prefix tables. In S. Albers and J.-Y. Marion, editors, *26th International Symposium on Theoretical Aspects of Computer Science (STACS 2009)*, pages 289–300, Dagstuhl, Germany, 2009. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany. http://drops.dagstuhl.de/opus/volltexte/2009/1825.
6. J.-P. Duval, T. Lecroq, and A. Lefebvre. Border array on bounded alphabet. *Journal of Automata, Languages and Combinatorics*, 10(1):51–60, 2005.

7. F. Franek, S. Gao, W. Lu, P. J. Ryan, W. F. Smyth, Y. Sun, and L. Yang. Verifying a Border array in linear time. *Journal on Combinatorial Mathematics and Combinatorial Computing*, 42:223–236, 2002.
8. F. Franek and W. F. Smyth. Reconstructing a Suffix Array. *International Journal of Foundations of Computer Science*, 17(6):1281–1295, 2006.
9. T. I, S. Inenaga, H. Bannai, and M. Takeda. Counting parameterized border arrays for a binary alphabet. In A. H. Dediu, A.-M. Ionescu, and C. Martín-Vide, editors, *LATA*, volume 5457 of *Lecture Notes in Computer Science*, pages 422–433. Springer, 2009.
10. Y. Li and W. F. Smyth. Computing the cover array in linear time. *Algorithmica*, 32(1):95–106, 2002.
11. M. Lothaire, editor. *Algebraic Combinatorics on Words*. Cambridge University Press, 2001.
12. M. Lothaire, editor. *Applied Combinatorics on Words*. Cambridge University Press, 2005.