Computing the Minimum k-Cover of a String

Richard Cole [1§], Costas S. Iliopoulos [2†], Manal Mohamed [2‡],
W. F. Smyth [3¶] and Lu Yang[4]

[1] Computer Science Department, Courant Institute of Mathematical Sciences,
New York University, New York, NY 10012-1185 U.S.A.
cole@cs.nyu.edu

[2] Algorithm Design Group, Department of Computer Science,
King's College London, London WC2R 2LS, England
{csi,manal}@dcs.kcl.ac.uk

[3] Algorithms Research Group, Department of Computing & Software,
McMaster University, Hamilton ON L8S 4K1, Canada &
School of Computing, Curtin University, Perth WA 6845, Australia
smyth@mcmaster.ca

[4] IBM Canada Limited, 8200 Warden Avenue, Markham ON L6G 1C7, Canada
luyang@ca.ibm.com

Abstract. We study the minimum k-cover problem. For a given string x of length n and an integer k, the minimum k-cover is the minimum set of k-substrings that covers x. We show that the on-line algorithm that has been proposed by Iliopoulos and Smyth [IS92] is not correct. We prove that the problem is in fact NP-hard. Furthermore, we propose two greedy algorithms that are implemented and tested on different kind of data.

Keywords: string algorithm, k-cover, data compression, NP-complete, greedy algorithm.

1 Introduction

The *minimum k-cover* problem is to compute, for a given string x and an integer $k < |x|$, a set $U = \{u_1, u_2, \ldots, u_m\}$ of substrings of x such that:

(i) every u_i is of length k;

(ii) the set U covers the string x;

(iii) the number $m = |U|$ of such substrings is the smallest possible.

§ Work supported in part by NSF grant CCR-0105678.
† Partially supported by a Marie Curie fellowship,Wellcome and Royal Society grants.
‡ Supported by an EPSRC studentship.
¶ Supported by a grant from the Natural Sciences & Engineering Research Council of Canada.

This problem was studied by Iliopoulos and Smyth [IS92], where they designed an $O(n^2(n-k))$ on-line algorithm. The idea of a k-cover is a generalization of the idea of a cover, where a string w is called a cover of a string x if x can be constructed by concatenations and superpositions of w. For example, if $x = ababaaba$, then aba and x are the covers of x. If $w \neq x$ covers x then w is called a *proper cover* of a *coverable* string x. The notion of a cover was introduced by Apostolico *et al.* [AFI91], where they gave a linear time algorithm for the shortest covers problem. Breslauer [B92] presented an on-line algorithm for the same problem. Moore and Smyth [MS94] presented a linear time algorithm to compute all the covers of every prefix of a string. An on-line algorithm for the same problem was developed by Li and Smyth [LS02]. Two $O(n \log n)$ algorithms for computing all maximal coverable substrings of a given string were also presented, one by Iliopoulos and Mouchard [IM93] and the other by Brodal and Pederson [BP00]. A lot of work has been done on parallel computation of covers; see for example [B94] and [IP94].

A minimum k-cover provides a theoretical classification of strings according to approximate periodicity. For every k, some strings have a minimum k-cover of cardinality 1, some a minimum k-cover of cardinality 2, and so on. Thus for a range of k, a minimum k-cover can provide a measure of how close to periodic every string x is. Practically, a minimum k-cover has a potential application in data compression of nonrandom strings. A minimum k-cover may also be useful in DNA sequence analysis. A DNA sequence is based on a four-letter *alphabet* for example $\{a, c, g, t\}$. Hence, finding the k-cover of a DNA sequence could be helpful for the analysis of its structure.

In this paper, we briefly present Iliopoulos and Smyth's on-line algorithm. Their algorithm computes the minimum k-covers for all prefixes of a given string x in $O(n^2(n-k))$ time. We show why the algorithm does not work correctly (Section 3). In the rest of the paper we consider two closely-related problems:

(Problem 1) for given x, k and m, *decide* whether there exists a k-cover of x of cardinality m;

(Problem 2) *compute* a minimum k-cover of x.

For $m = 1$, Problem 1 can be solved in $\Theta(n)$ time simply by computing all the covers of x [MS94, MS95, LS02] while at the same time testing to determine whether or not each one is of length k. For $m > 1$ we show by reduction to 3-SAT that Problem 1 is NP-hard (Section 4). We then describe two efficient algorithms that yield approximate solutions to Problem 2 (Section 5). These approximation algorithms have been tested and shown to provide good results (Section 6). More approximation algorithms were proposed in [Y00].

2 Preliminaries

A *string* is a sequence of zero or more symbols drawn from an alphabet Σ. The set of all strings over Σ is denoted by Σ^*. The string of length zero is the *empty string* ϵ; a string x of length $n > 0$ is represented by $x_1x_2\cdots x_n$, where $x_i \in \Sigma$ for $1 \leq i \leq n$. A string w is a *substring* of x if $x = uwv$ for $u, v \in \Sigma^*$. More precisely, let $i \leq n$ and $j \leq n$ denote nonnegative integers: if $1 \leq i \leq j$, $x[i..j]$ denotes the substring of x

that starts at position i and has length $j-i+1$; otherwise, $x[i..j]=\epsilon$. A string w is a *prefix* of x if $x=wu$ for some $u \in \Sigma^*$. Similarly, w is a *suffix* of x if $x=uw$ for some $u \in \Sigma^*$.

The string xy is a *concatenation* of two strings x and y. The concatenation of k copies of x is denoted by x^k. For two strings $x=x_1\cdots x_n$ and $y=y_1\cdots y_m$ such that $x_{n-i+1}\cdots x_n = y_1\cdots y_i$ for some $i\geq 1$ (that is, such that x has a suffix equal to a prefix of y), the string $x_1\cdots x_n y_{i+1}\cdots y_m$ is said to be a *superposition* of x and y. Alternatively, we may say that x *overlaps* with y.

A substring w is said to be a *cover* of a given string x if every position of x lies within an occurrence of a string w within x. Additionally, if $|w|<|x|$ then w is called a *proper cover* of x. For example, x is always a cover of x, and $w=aba$ is a proper cover of $x=abaababa$.

For a given a nonempty string x of length n and a set

$$U=\{u_1,u_2,\ldots,u_m\}$$

of m strings each of length k, we say that U is a *k-cover* of x if and only if every position of x lies within an occurrence of some u_i, $1\leq i\leq m$. If m is the minimum integer for which such a set U exists, then U is said to be a *minimum k-cover* of x. To avoid trivialities we suppose throughout that $1<k<n/2$. Note that $1\leq m\leq \lceil n/k\rceil$. Next we state some basic facts about the minimum k-cover.

Fact 1 The prefix $x[1..k]$ and the suffix $x[n-k+1..n]$ are both necessarily elements of every minimum k-cover of x.

Fact 2 The cardinality of a minimum k-cover of a string of length n is at most $\lceil n/k\rceil$.

Fact 3 A minimum k-cover of a string x is not unique.

For example, if $x=abcdefg$, then the sets

$$\{abc,bcd,efg\},\{abc,cde,efg\},\{abc,def,efg\}$$

are all minimum 3-covers of x.

In [IS92], the number of distinct minimum k-covers of a given string x of length n has been proved to be exponential in n. This is a major complicating factor in the design of polynomial time algorithm for computing the minimum k-covers of a given string.

3 Iliopoulos & Smyth On-Line Algorithm

Recall that in [IS92], Iliopoulos and Smyth designed an $O(n^2(n-k))$ time on-line algorithm for computing a minimum k-cover of a given string x of length n. Their algorithm scans a given string x from left to right and iteratively calculates a minimum k-cover for every prefix of x. The algorithm is based upon the following two main ideas:

1. Fact 1 states that a minimum k-cover of $x[1..i+1]$ must include the suffix $x[i-k+2..i+1]$. This is used as a yardstick to find a minimum k-cover.

2. For $i \geq k$, a minimum k-cover of $x[1..i+1]$ depends only on the minimum k-covers of the previous k positions; that is, the minimum k-cover of $x[1..i-k+1], \ldots, x[1..i-1], x[1..i]$.

To achieve efficiency, the algorithm stores for each positions i in x an array which identifies all the k-substrings that occur in *at least one* of the minimum k-covers. Let c_i be the cardinality of this set. At step $i+1$, the algorithm checks for each position $j \in i-k+1..i$, whether the current suffix $x[i-k+2..i+1]$ has already been included in the stored minimum k-cover of $x[1..j]$. If so then the set covers $x[1..i+1]$, otherwise the current suffix has to be added to the set. Among these k candidates, the algorithm chooses a set with the smallest cardinality as a minimum k-cover of $x[1..i+1]$. For more details see [IS92].

Lemma 3.1 For $i \geq 2k$ and $l, l' = 1, 2, \ldots$, let $U_{i,l}$ denotes the distinct minimum k-cover for $x[1..i]$. Then every minimum set $U_{i+1,l}$ is a superset of some minimum set $U_{j,l'}$, $i-k+1 \leq j \leq i$.

The above lemma is stated in [IS92] and it follows directly from the two ideas stated at the beginning of this section. The algorithm as we briefly described also relies on the correctness of the lemma. In the next example we will show that the lemma is not correct and consequentially nor is the algorithm. The following example illustrates just one of the situations where the algorithm fails to compute a minimum k-cover.

Example: If $x = bacaababbaaaccaabbabbbaaaac$ and $k = 3$ then when $i+1 = 27$, $j \in 24..26$, and position 27 should form its minimum k-cover from position 24 because $c_{24} = \min(c_j),\ j \in 24..27$. The minimum k-covers of position 24 are as follows:

$$U_{24,1} = \{bac, aab, abb, baa, cca\},$$

$$U_{24,2} = \{bac, aab, abb, baa, acc\}.$$

Neither of them contains the suffix aac, so we get $c_{27} = c_{24} + 1 = 6$, and accordingly the minimum k-covers of position 27 are as follows:

$$U_{27,1} = \{bac, aab, abb, baa, cca, aac\},$$

$$U_{27,2} = \{bac, aab, abb, baa, acc, aac\}.$$

But we can find at least one minimum k-cover that is different from $U_{27,1}$ and $U_{27,2}$; namely:

$$U_{27,3} = \{bac, aab, abb, baa, caa, aac\}.$$

$U_{27,3}$ is a k-cover of position 24, but not the minimum. However it will contribute to the minimum when position 27 is reached. There is a potential problem for future calculations if we lose $U_{27,3}$ at position 27; for example if we extend x by adding aa to the end. As we can see, $U_{27,3}$ can be a minimum k-cover of $x[1..29]$. Without keeping $U_{27,3}$, we shall get $c_{29} = 7$, one greater than the minimum.

The above suggests that in order to compute a minimum k-cover of the current position, we have to refer to every single k-cover of the previous positions. Since the number of minimum k-covers of a string may be exponential, we doubt that the problem of computing a minimum k-cover can be solved in polynomial time.

4 Problem 1 and NP-Completeness

The k-cover problem is to find a set cover of minimum size for a given string. Restating this optimization problem as a decision one, we wish to determine whether a given string has a k-cover of a given size m.
k_m-COVER $= \{\langle x, k, m\rangle$: string x has a k-cover of size $m\}$.

The following theorem shows that this problem is NP-complete.

Theorem 4.1 The k_m-COVER $\in$ NP.

Proof. To show that k_m-COVER $\in$ NP, for a given string x, we use the set U_m of m substrings all of length k as a certificate for x. Checking whether U_m is a k-cover can be accomplished in $O(n \log n)$ time by checking whether, for each position $1 \leq i \leq n$, i is covered by at least one of the k-substrings in U_m.

We next prove that 3-SAT $\leq_p$ k_m-COVER, which shows that a minimum k-cover problem is NP-hard. 3-SAT is well-known to be NP-complete [C71]. We transform 3-SAT to k_m-COVER. Let $V = \{v_1, v_2, \ldots, v_p\}$ be a set of variables, $C = \{c_1, c_2, \ldots, c_q\}$ be the set of clauses and $F = c_1 \wedge c_2 \wedge \ldots \wedge c_q$ be a 3-SAT formula with $c_i = \ell_1^i \vee \ell_2^i \vee \ell_3^i$, $1 \leq i \leq q$.

We shall show how to construct from F a string x such that x will have a k-cover of size m if and only if F is satisfiable. We choose $k = 3$ and note that there is an easy reduction to 2-CNF for $k = 2$. The string x is build of substrings separated by sequences of $sssss$; hence sss is one of the chosen covering k-strings, and thus we can focus on the individual substrings. The construction will be made up of truth-setting components, and satisfaction testing components.

Variable Choice

For each variable $v \in V$, we construct the following 6 substrings (each substring is proceeded and followed by $sssss$); each character is indexed by v:

(i) $\#_a\ r\ r\ \$\ v\ \phi\ \pi\ r\ r\ \#_a$	**(ii)**$\#_b\ t\ t\ \$\ \bar{v}\ \phi\ \pi\ t\ t\ \#_b$
(iii)$\#_a$	**(iv)** $\#_b$
(v)$\#_a\ \#_b$	**(vi)**$\#_b\ \#_a$

The only ways to cover the above strings with 9 or fewer length 3 strings, are one of the following (notice the uninteresting flexibility in (v) and (vi)):

1. $\{ss\#_a, rr\$, v\phi\pi, rr\#_a, \#_b tt, \$\bar{v}\phi, \pi tt, \#_b ss\}$ and one of $\{s\#_b\#_a, \#_b\#_a s\}$.
2. $\{\#_a rr, \$v\phi, \pi rr, \#_a ss, ss\#_b, tt\$, \bar{v}\phi\pi, tt\#_b\}$ and one of $\{s\#_a\#_b, \#_a\#_b s\}$.

To see this, consider covering string (iii). It can be done by one of $ss\#_a$, $\#_a ss$, $s\#_a s$, but only the first two could be used elsewhere, so one of them may as well be chosen. Clearly, 8 strings at least are needed to cover (i) and (ii) as they have no length 3 substring in common. Thus, to use only 1 additional string to cover (v) and (vi) we need to choose either $ss\#_a, \#_b ss$ or $\#_a ss, ss\#_b$.

The choice $v\phi\pi$ and $\$\bar{v}\phi$ (given by choosing $ss\#_a$) corresponds to $v = T$ while the choice $\bar{v}\phi\pi$ and $\$v\phi$ (given by choosing $\#_a ss$) corresponds to $v = F$.

Clause Satisfiability

For each clause $c \in C$, where $c = \ell_1 \vee \ell_2 \vee \ell_3$, the following substrings are created, again preceded and followed by $sssss$. The characters, except for $\$_i, \phi_i, \pi_i, \ell_i, i = 1, 2, 3$ are indexed by c also; $\$_i, \phi_i, \pi_i, \ell_i$ carry the index for the literal.

(i)$\$_1\ \ell_1\ \phi_1\ \pi_1\ h_1$	**(ii)** $\$_2\ \ell_2\ \phi_2\ \pi_2\ h_2$	**(iii)** $\$_3\ \ell_3\ \phi_3\ \pi_3\ h_3$
(iv)$\$_1$	**(v)**$\$_2$	**(vi)**$\$_3$
(vii)h_1	**(viii)**h_2	**(ix)**h_3
(x)$\phi_1\ \pi_1\ h_1\ d_1\ \phi_2\ \pi_2\ h_2$	**(xi)**$\phi_2\ \pi_2\ h_2\ d_2\ \phi_3\ \pi_3\ h_3$	**(xii)**$\phi_3\ \pi_3\ h_3\ d_3\ \phi_1\ \pi_1\ h_1$
(xiii)ϕ_1	**(xiv)**ϕ_2	**(xv)**ϕ_3

To cover (iv)-(ix) and (xiii)-(xv) we may as well choose $ss\$_i$, $h_i ss$ and $ss\phi_i$ as these are the only reusable substrings.

If ℓ_i is true, then $\ell_i\phi_i\pi_i$ was already chosen; otherwise $\$_i\ell_i\phi_i$ was chosen. Thus, if ℓ_i is false; in (i)-(iii), π_i remains to be covered. The only reusable covering string is $\phi_i\pi_i h_i$.

Consider strings (x)-(xii) and suppose at least one ℓ_i is true. Without loss of generality let it be ℓ_1. Then it is not hard to see that 5 more strings that include $\phi_2\pi_2h_2$ and $\phi_3\pi_3h_3$ thereby covering π_2 in (ii) and π_3 in (iii) suffice. We choose: $\phi_2\pi_2h_2, \phi_3\pi_3h_3, \pi_1h_1d_1, d_2\phi_3\pi_3$ and $d_3\phi_1\pi_1$. It is not hard to see that 5 covering strings are needed: 3 to cover d_1, d_2 and d_3, but this can only completely cover one of π_1, π_2 and π_3 as each occurs twice, and hence two more covering strings are needed for the remaining pair among π_1, π_2 and π_3.

If no ℓ_i is true, we are obliged to choose $\phi_1\pi_1h_1, \phi_2\pi_2h_2$ and $\phi_3\pi_3h_3$ as well as 3 strings to cover d_1, d_2 and d_3. At least 6 covering strings in all are needed. Thus, if F is satisfiable then the full string can be covered by

$$m = 9p + 6p + 3q + 5q + 1 = 15p + 8q + 1$$

covering strings, where p is the number of variables in F and q is the number of clauses. Otherwise, it needs at least $15p + 8q + 2$ covering strings. □

5 Approximate Minimum k-Cover

In this section we introduce two greedy algorithms to compute a minimum k-cover. The greedy method works by picking, at each stage, the k-substring which covers the greatest number of uncovered positions. The first algorithm works globally while the second algorithm follows a local strategy. To calculate all possible k-substrings in a given string x, both greedy algorithms use Crochemore's partitioning algorithm [C81] to preprocess the input string x.

Originally, Crochemore's algorithm was designed to compute the *repetitions* in a string in $O(n \log n)$ time. A string has a *repetition* when it has at least two consecutive equal substrings. For example, $abab$ is a repetition in $aababba = a(ab)^2ba$. We shall use the algorithm in another way — to find the sets of the starting positions of all the distinct substrings of length k in a given string x. This idea can be expressed more precisely as follows:

Given a string $x[1..n]$ and an integer k, Crochemore's algorithm is used to compute the equivalence classes of all equal substrings of length k in x. We denote these equivalence classes by $e_1, e_2, ..., e_m$, where the elements in e_i are sorted integers denoting starting positions of equal substrings, and m is the number of possible equivalence classes returned by the algorithm.

These elements are stored using a global array $L[1..n]$, such that $L[i]$ is the next position in the same equivalence class of equal substrings of length k. That is, $L[i] = j$ if $L[i..i+k-1] = x[j..j+k-1]$ and the circular sequence $i, L[i], L[L[i]], \ldots, L^{\ell}[i] = i$ identifies all ℓ k-substrings in x that are equal to $x[i..i+k-1]$.

For example, if $x = abaababaabaab$ and $k = 3$ then $e_1 = \{3, 8, 11\}$, $e_2 = \{1, 4, 6, 9\}$, $e_3 = \{2, 7, 10\}$, and $e_4 = \{5\}$ are the equivalence classes. Where aab, aba, baa, bab are the corresponding 3-substrings. Hence, the value of array L is as follows:

	1	2	3	4	5	6	7	8	9	10	11	12	13
$x =$	a	b	a	a	b	a	b	a	a	b	a	a	b
$L[i]$	4	7	8	6	5	9	10	11	1	2	3		
$Eid[i]$	2	3	1	2	4	2	3	1	2	3	1		

In the above, $Eid[i]$ identifies the equivalence class containing position i. In the following subsections, we shall present two approximation algorithms. We call the first Global-Uncovered and the second Local-Uncovered.

5.1 Global-Uncovered Algorithm

Recall that the greedy algorithm works by selecting one k-substring at a time that covers the most positions among the uncovered ones. Our greedy algorithm is comparable to the greedy one [J74] to construct the minimum set cover. The cost of a greedy solution is known to come always within a multiplicative factor of $\mathcal{H}(\max_j |EC_j|)$, where EC_j is the number of positions that could be covered by the k-substring j. Here, $\mathcal{H}(d) = \sum_{i=1}^{d} \frac{1}{i}$ is the dth harmonic number and is bounded by $1 + \log d$. This was shown by Johnson [J74] and Lovasz [L75] for the general SET COVER problem.

The key to Algorithm Global-Uncovered is finding the equivalence class which can cover the maximum number of so-far-uncovered positions efficiently. The details of the algorithm are provided in Figure 1. To achieve efficiency, the algorithm uses the following data structures:

1. An array $Ebucket[1..n]$ indexed by the number of so-far-uncovered positions that could be covered by a single equivalence class. Each element (bucket) of the array is doubly-linked list of the equivalence classes that could cover equal number of so-far-uncovered positions. Thus, every element of the doubly linked list contains an index of an equivalence class in addition to the *left* and the *right* pointers to the adjacent elements.

2. A two dimensional array $Eptr[1..m]$ indexed by the equivalence class j. Where $Eptr[j][bucket]$ identifies the bucket that includes j in its doubly linked list. In other words, equivalence class j could cover $Eptr[j][bucket]$ so-far-uncovered positions. Additionally $Eptr[j][ptr]$ is a pointer to the corresponding element of the doubly linked list $Ebucket[Eptr[j][bucket]]$. Thus, any elements of the doubly linked lists can be referenced in constant time by using $Eptr$.

```
Algorithm Global-Uncovered(x, k)
Input: A string x of length n, an integer 0 < k < n
Output: An approximate minimum k-cover U_g
(L[1..n], Eid[1..n], start[1..m], m) ←CrochemorePar(x, k)
cover_so_far[1..n] ←F, F, ..., F
initialization:
U_g ←∅
for e ←1 to m do
        Euncov[e] ←0  **number of positions that could be covered by equivalence class e**
for i←1 to n − k + 1
        if i < L[i]
          then Euncov[Eid[i]] += min(k, L[i] − i)
          else Euncov[Eid[i]] += k
(Ebuckect, Eptr) ←Bucket-Sort(Euncov)
The algorithm:
k_prefix, k_suffix ←Eid[1], Eid[n − k + 1]
GU-Cover(k_prefix, Ebucket, Eptr)
Add(U_g, k_prefix)
if k_suffix ≠ k_prefix
   then GU-Cover(k_suffix, Ebucket, Eptr)
        Add(U_g, k_suffix)
e ←Head(Ebucket)
while e ≠ 0
        GU-Cover(e, Ebucket, Eptr)
        Add(U_g, e)
        e ←Head(Ebucket)
return U_g


Function GU-Cover(e, Ebucket, Eptr)
i ←start[e]   **the first element in the equivalence class e**
repeat
     for j ←1 to k do
            if cover_so_far[i + j − 1] = F then
                    cover_so_far[i + j − 1]←T
                    for every  l ∈ Eid[(i + j − 1) − k + 1], ... Eid[i + j − 1] do
                            Delete(Ebucket[Eptr[l][bucket]],Eptr[l][ptr])
                            if Eptr[l][bucket] ≠ 1
                               then Insert(Ebucket[Eptr[l][bucket − 1]],Eptr[l][ptr])
                            Eptr[l][bucket] ←Eptr[l][bucket] − 1
     i ←L[i]
until (i = start[e])
```

Figure 1: Global-Uncovered Algorithm.

Once *Ebucket* is established, the k-prefix and the k-suffix are the first elements to be included in the approximate minimum k-cover. The algorithm then iteratively choose a head element of *Ebucket* as an element of the approximate minimum k-cover. The head element is an equivalence class that covers the largest number of so far uncovered positions. Finding such equivalence classes costs $O(n)$ time throughout the calculations.

The algorithm requires $O(n \log n)$ time to run Crochemore's algorithm and an additional $O(n)$ time to construct and initialize *Ebucket* and *Eptr*. Note that a linear time Bucket-Sort has been used because the number of positions that could be covered by any equivalence class is bounded.

For each position i, $cover\_so\_far[i]$ is initialized to F and set to T once during the calculation. When $cover\_so\_far[i]$ is set from F to T, $O(k)$ elements in *Ebucket* may need to be deleted from the current bucket and inserted to the next bucket. Each rearrangement costs $O(1)$ time. Thus, the total time required to maintain the elements in *Ebucket* throughout the calculation is $O(kn)$. Summing the above gives the total running time: $O(n \log n) + O(n) + O(kn) = \max\{O(n \log n), O(kn)\}$ time, which for a fixed k, asymptotically approaches $O(n \log n)$ as n increases to ∞.

5.2 Local-Uncovered Algorithm

Algorithm Local-Uncovered chooses its candidate element, of the approximate minimum k-cover, in a range of $Eid[left\_uncover - k + 1]..Eid[left\_uncover]$; the integer $left\_uncover$ keeps track of the leftmost so-far-uncovered position. The algorithm uses the array $uncover\_no$. The array $uncover\_no[1..m]$ is indexed by the equivalence classes, where $uncover\_no[j]$ is the number of positions corresponding to equivalence class j that have not been covered. Hence, the values of the array need to be updated dynamically during the computation. The details of the algorithm are provided in Figure 2.

The initialization is just the same as in Global-Uncovered. However, we need to update $uncover\_no$. As in Global-Uncovered, the k-prefix and the k-suffix are the first two elements to be included in the approximate minimum k-cover. The algorithm then tries to cover the leftmost uncovered position with the k-substring corresponding to the equivalence class which can cover the maximum number of uncovered positions. That is, let $j = left\_uncover$ if $j < n$, then the chosen k-substring is the one corresponding to equivalence class satisfying

$$max\{uncover\_no[Eid[j-k+1], uncover\_no[j-k+2], \ldots, uncover\_no[Eid[j]]\}.$$

A brief analysis of the algorithm shows that the algorithm requires:

- $O(n \log n)$: to run Crochemore's algorithm;
- $O(n)$: Step 2, the loop on (Steps 6-9), and the total time spent in Add();
- $O(k)$: the loop on (Steps 19-23);
- $O(kn)$: is the total time of the LU-Cover subroutine.

Summing the above gives the total running time $O(n \log n) + O(n) + O(k) + O(kn) = max\{O(n \log n), O(kn)\}$ time.

```
Algorithm Local-Uncovered(x, k)
Input: A string x of length n, an integer 0 < k < n
Output: An approximate minimum k-cover U_l
(L[1..n], Eid[1..n], m) ←CrochemorePar(x, k)
cover_so_far[1..n] ←F, F, ..., F
initialization:
U_l ←∅
left_uncover ←1
for i←1 to n − k + 1 do
        if i < L[i]
          then uncover_no[Eid[i]] += min(k, L[i] − i)
          else uncover_no[Eid[i]] += k
The algorithm:
k_prefix, k_suffix ←Eid[1], Eid[n − k + 1]
LU-Cover(k_prefix, 1, uncover_no, left_uncover)
Add(U_l, k_prefix)
if k_suffix ≠ k_prefix then
        LU-Cover(k_suffix, n − k + 1, uncover_no, left_uncover)
        Add(U_l, k_suffix)
while left_uncover < n do
        max = 0
        for j ←1 to k do
                if uncover_no[Eid[left_uncover − j + 1]] > max then
                        max ←uncover_no[Eid[left_uncover − j + 1]]
                        e ←Eid[left_uncover − j + 1]
                        s ←left_uncover − j + 1
        LU-Cover(e, s, uncover_no, left_uncover)
        Add(U_l, e)
return U_l


Function LU-Cover(e, start, uncover_no, left_uncover)
i ←start
repeat
    for j ←1 to k do
            if cover_so_far[i + j − 1] = F then
                    cover_so_far[i + j − 1]←T
                    for every l ∈ Eid[(i + j − 1) − k + 1], ... Eid[i + j − 1] do
                            uncover_no[l] −= 1
    i ←L[i]
until (i = start)
while left_uncover ≤ n and cover_so_far[left_uncover] do
        left_uncover++
```

Figure 2: Local-Uncovered Algorithm.

Length	$\|U_N\|$	$\|U_{GU}\|$	$\|U_{LU}\|$	$\|U_{best}\|$	α_N (%)	α_{GU} (%)	α_{LU} (%)
100	12	11	11	11	9.09	0	0
200	14	14	14	14	0	0	0
300	14	15	15	14	0	7.14	7.14
400	16	15	17	15	6.67	0	13.3
500	17	17	17	17	0	0	0
600	16	16	16	16	0	0	0
700	18	16	16	16	12.5	0	0
800	17	17	19	17	0	0	11.8
900	18	16	18	16	12.5	0	12.5
1000	18	17	16	16	12.5	6.25	0
Average (%)	/	/	/	/	5.33	1.34	4.47

Table 1: Pseudo-Random Strings on Alphabet $\{a,\ b,\ c\}$, and $k = 3$

6 Experimental Results

We used four types of strings: sturmian strings, pseudo random strings on the alphabets: $\{a, b\}$, $\{a, b, c\}$, $\{a, b, c, d\}$, DNA sequences*, and English text. In order to compare our approximate methods in term of effectiveness, we developed a naive algorithm based on the Iliopoulos and Smyth algorithm. This naive algorithm finds the minimum k-cover at position $i + 1$ by testing each position $j \in i - k + 1..i$ in the same way as in Iliopoulos and Smyth's. However, the key difference is that the algorithm stores not only the covers that are minimum but also those that are one more than minimum at every position. Thus, the aim here is to store as much information as possible taking into consideration the limitation of the computer's resources. The implementation results show that the naive algorithm does not always yield the best k-cover - in most cases the two approximate algorithms yield better results. Let U_{min} be the minimum k-cover of a string x, U_N be the result computed by our naive method, U_{GU} be the result computed by Global-Uncovered algorithm, and U_{LU} be the result computed by Local-Uncovered algorithm. Then the following simplifying assumption has been made:

$$|U_{min}| \leq |U_{best}| = min\{|U_N|, |U_{GU}|, |U_{LU}|\}$$

Table 1, 2, 3 show that Algorithm Global-Uncovered yields the best result in most cases, the naive algorithm never exceed a deviation of 7.83%, and Algorithm Local-Uncovered never exceed 6.24%. The following observations are also worth mentioning:

- The Sturmian strings are very well-structured. For the tested Sturmian strings, from length of 20 to 1000, for every $k \in 3, 4, 5$, $|U_{best}| = 2$.
- For the tested pseudo-random strings and DNA sequences, $|U_{best}|$ increases as the values of k, the length n, and the alphabet size are increasing.
- Let $|U_{best-DNA}|$ denotes the cardinality of the approximate minimum k-cover of DNA sequence and $|U_{best-abcd}|$ denotes the cardinality of the approximate

*excerpted from www.cbs.dtu.dk/databases/DNA2protSS/nucall.seq.

Length	$\|U_N\|$	$\|U_{GU}\|$	$\|U_{LU}\|$	$\|U_{best}\|$	α_N (%)	α_{GU} (%)	α_{LU} (%)
100	19	19	19	19	0	0	0
200	25	26	27	25	0	4.00	8.00
300	32	29	29	29	10.3	0	0
400	37	34	36	34	8.80	0	5.88
500	36	36	35	35	2.86	2.86	0
600	37	36	37	36	2.78	0	2.78
700	37	35	38	35	5.71	0	8.57
800	42	37	39	37	16.2	0	5.41
900	42	35	42	35	20	0	20
1000	42	38	39	38	10.5	0	2.63
Average (%)	/	/	/	/	7.71	0.68	5.32

Table 2: Pseudo-Random Strings on Alphabet $\{a,\ b,\ c,\ d\}$, and $k = 3$

Length	$\|U_N\|$	$\|U_{GU}\|$	$\|U_{LU}\|$	$\|U_{best}\|$	α_N (%)	α_{GU} (%)	α_{LU} (%)
60	13	13	13	13	0	0	0
126	21	22	23	21	0	4.76	9.52
171	23	22	23	22	4.54	0	4.54
234	25	24	26	24	4.17	0	8.33
312	32	29	30	29	10.3	0	3.45
432	26	27	29	26	0	3.85	11.5
591	34	31	35	31	9.68	0	12.9
771	40	34	36	34	17.6	0	5.89
1233	43	38	37	37	24.3	2.70	0
Average (%)	/	/	/	/	7.83	1.26	6.24

Table 3: DNA Sequences, and $k = 3$

minimum k-cover of pseudo-random strings on alphabet $\{a,\ b,\ c,\ d\}$. For the same value of k and n, $|U_{best-DNA}| < |U_{best-abcd}|$. We can make a conjecture that DNA sequences are better structured than pseudo-random strings on an alphabet of size 4.

Conclusions

We have shown that for $k \geq 2$, the k-cover problem (Problem1) is NP-Complete. We have then proposed two $O(n \log n)$ greedy algorithms that can be used to calculate an approximate minimum k-cover. The results obtained by the algorithms are believed to come within a multiplicative factor of the minimum. Prove this has been left as an open problem.

References

[AFI91] A. Apostolico, M. Farach & C. S. Iliopoulos, **Optimal superprimitivity testing for strings**, *Information Processing Letters 39-1* (1991) 17-20.

[B92] D. Breslauer, **An on-line string superprimitivity test**, *Information Processing Letters 44* (1992) 345-347.

[B94] D. Breslauer, **Testing string superprimitivity in parallel**, *Information Processing Letters 49-5* (1994) 235-241.

[BP00] G. S. Brodal & C. Pederson, **Finding maximal quasiperiodicities in strings**. *In Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching (CPM)* (2000) 397-411.

[C71] Stephen A. Cook, **The complexity of theorem-proving procedures**, *Proc. Third Annual ACM Symp. on Theory of Computing* (1971) 151-158.

[C81] M. Crochemore, **An optimal algorithm for computing all the repetitions in a word**, *Information Processing Letters 12-5* (1981) 244-248.

[IM93] C. S. liopoulos & L. Mouchard, **An $O(n \log n)$ algorithm for computing all maximal quasiperiodicities in strings**, *Theoratical Computer Science 119-2* (1993) 247-265.

[IP94] C. S. Iliopoulos & K. Park, **An optimal $O(\log \log n)$-time algorithm for parallel superprimitivity testing**, *Journal of the Korea Information Science Society 21-8* (1994) 1400-1404.

[IS92] C. S. Iliopoulos & W. F. Smyth, **An on-line algorithm of computing a minimum set of k-covers of a string**, *Proc. Ninth Australasian Workshop on Combinatorial Algorithms (AWOCA)*, (1998) 97-106.

[J74] D. S. Johnson, **Approximation algorithms for combinatorial problems**, *Journal of Computer and System Science 9* (1974) 256-278.

[MS94] D. Moore & W. F. Smyth, **An optimal algorithm to compute all the covers of a string**, *Information Processing Letters 50-5* (1994) 239-246.

[MS95] D. Moore & W. F. Smyth, **A correction to: An optimal algorithm to compute all the covers of a string**, *Information Processing Letters 54* (1995) 101-103.

[L75] L. Lovasz, **On the ratio of optimal integral and fractional covers**, *Discrete Mathematics 13* (1975) 383-390.

[LS02] Y. Li & W. F. Smyth, **Computing the cover array in linear time**, *Algorithmica 32-1*, (2002) 95-106.

[Y00] Lu Yang, **Computing the Minimum k-Cover of a String**, *M. Sc. thesis, McMaster University*, (2000).