Computing All Approximate Enhanced Covers with the Hamming Distance

Ondřej Guth

Department of Theoretical Computer Science
Faculty of Information Technology
Czech Technical University in Prague
ondrej.guth@fit.cvut.cz

Abstract. A border $\boldsymbol{p}$ of a string $\boldsymbol{x}$ is an enhanced cover of $\boldsymbol{x}$ if the number of positions of $\boldsymbol{x}$ that lie within some occurrence of $\boldsymbol{p}$ is the maximum among all borders of $\boldsymbol{x}$. In this paper, more general notion based on the enhanced cover is introduced: a k-approximate enhanced cover, where fixed maximum number of errors k in the Hamming distance is considered. The k-approximate enhanced cover of $\boldsymbol{x}$ is its border and its k-approximate occurrences are also considered in the covered number of positions of $\boldsymbol{x}$. An $\mathcal{O}(n^2)$-time and a $\mathcal{O}(n)$-space algorithm that computes all k-approximate enhanced covers of a string of length n is presented.

Keywords: string regularity, approximate cover, enhanced cover, quasiperiodicity, suffix automaton, Hamming distance, border

1 Introduction

Searching repetitive structures of strings, so-called regularities of strings, has been intensively studied for many years in many fields of computer science, e.g., combinatorics on strings, pattern matching, data compression and molecular biology, and many related notions have been introduced: periods, squares, covers, seeds, etc. [10] Those long-time known repetitive structures provide compact description of a string. However, they are quite restrictive and it is rare that an arbitrary string has a non-trivial regularity of that kind (e.g., not every string has a cover shorter than itself). Therefore, there have been attempts to introduce more relaxed repetitive structures, e.g., their approximate versions. Quite recently, a term of *enhanced cover* [3] has been introduced; every string having a (non-empty) border has also an enhanced cover.

In order to provide even more general notion, the enhanced cover is extended to its approximate version in this paper, see Fig. 1. The problem of computing all k-approximate enhanced covers of a string with fixed maximum Hamming distance is solved using finite automata in a way which is easy to implement and understand and also consistent with finite automata based algorithms for similar problems. There is no other known solution of this problem.

This paper is organised as follows. Section 1.1 contains definitions of terms used through the text and also definition of the problem solved in this paper; in the section, previous and related work is also summarized. In Section 2, the algorithm solving the stated problem is presented; it starts with a description of a basic idea found in another paper, the algorithm is then described in words and also its pseudocode is shown; the time and space complexity is then stated and proved. In Section 3, a behaviour of an implementation of the presented algorithm is shown, depending on various input parameters.

Ondřej Guth: Computing All Approximate Enhanced Covers with the Hamming Distance, pp. 146–157.
Proceedings of PSC 2016, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05996-8

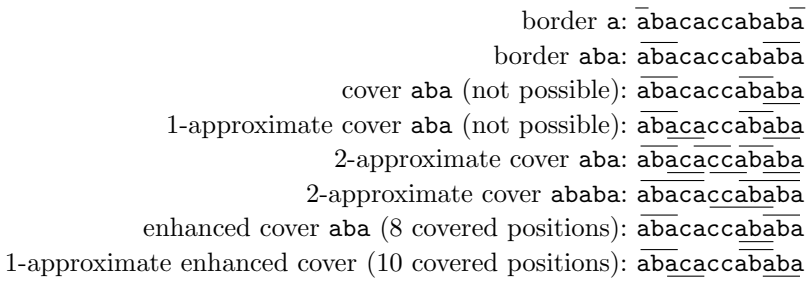

Figure 1. Regularities of the string $\boldsymbol{x} = \texttt{abacaccababa}$

1.1 Preliminaries

An *alphabet* is a finite set of symbols, denoted by A. A *string* $\boldsymbol{x}$ over alphabet A is a finite sequence of symbols of A, denoted by $\boldsymbol{x} \in A^*$. An empty string is denoted by ε. An i-th symbol of a string $\boldsymbol{x}$ is denoted by $\boldsymbol{x}[i]$, i.e., the first symbol of $\boldsymbol{x}$ is denoted by $\boldsymbol{x}[1]$. A substring of $\boldsymbol{x}$ starting at an i-th and ending at an j-th symbol is denoted by $\boldsymbol{x}[i..j]$, i.e., $\boldsymbol{x} = \boldsymbol{x}[1..|\boldsymbol{x}|]$. Assuming strings $\boldsymbol{p}, \boldsymbol{s}, \boldsymbol{u}, \boldsymbol{x} \in A^*$, where $\boldsymbol{x} = \boldsymbol{pus}$, the string $\boldsymbol{p}$ is a *prefix* of $\boldsymbol{x}$, the string $\boldsymbol{s}$ is a *suffix* of $\boldsymbol{x}$, and the string $\boldsymbol{u}$ is a *factor* (also known as a substring) of $\boldsymbol{x}$. An editing operation *replace* in a string $\boldsymbol{x} \in A^*$ is replacing a symbol $\boldsymbol{x}[i]$ with another symbol of A. Assuming strings $\boldsymbol{x}, \boldsymbol{y} \in A^*$ such that $|\boldsymbol{x}| = |\boldsymbol{y}|$, the *Hamming distance* of the strings $\boldsymbol{x}, \boldsymbol{y}$, denoted by $\mathsf{H}(\boldsymbol{x}, \boldsymbol{y})$, is the minimum number of operations replace necessary to convert $\boldsymbol{x}$ to $\boldsymbol{y}$. Assuming strings $\boldsymbol{p}, \boldsymbol{s}, \boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w} \in A^*$ and an integer $k \geq 0$, the string $\boldsymbol{v}$ is a *k-approximate factor* of the string $\boldsymbol{w}$ if $\boldsymbol{w}$ may be written as $\boldsymbol{pus}$ and $\mathsf{H}(\boldsymbol{u}, \boldsymbol{v}) \leq k$. The string $\boldsymbol{u}$ *has an occurrence* in the string $\boldsymbol{w}$ if $\boldsymbol{u}$ is a factor of $\boldsymbol{w}$. A factor $\boldsymbol{u}$ of $\boldsymbol{w}$ *occurs at position i* (that is also called an *end position*) in the string $\boldsymbol{w}$ if for all $j \in \{1, \ldots, |\boldsymbol{u}|\}$ it holds that $\boldsymbol{u}[j] = \boldsymbol{w}[i - |\boldsymbol{u}| + j]$. A position l of a string $\boldsymbol{w}$ *lies within some occurrence* of $\boldsymbol{u}$ in $\boldsymbol{w}$ if $\boldsymbol{u}$ occurs at a position i in $\boldsymbol{w}$ and $i - |\boldsymbol{u}| < l \leq i$. A k-approximate factor $\boldsymbol{v}$ of $\boldsymbol{w}$ *k-approximately occurs at position i* (that is also called a *k-approximate end position*) if there exists a factor $\boldsymbol{u}$ of $\boldsymbol{w}$ that occurs at the position i in $\boldsymbol{w}$ and $\mathsf{H}(\boldsymbol{u}, \boldsymbol{v}) \leq k$. A position l of a string $\boldsymbol{w}$ *lies within some k-approximate occurrence* of $\boldsymbol{v}$ in $\boldsymbol{w}$ if $\boldsymbol{v}$ k-approximately occurs at a position i in $\boldsymbol{w}$ and $i - |\boldsymbol{v}| < l \leq i$.

A *border* of a string $\boldsymbol{x}$ is simultaneously a prefix and a suffix of $\boldsymbol{x}$. A string $\boldsymbol{w}$ is a *cover* of $\boldsymbol{x}$ if every position of $\boldsymbol{x}$ lies within some occurrence of $\boldsymbol{w}$ in $\boldsymbol{x}$. A $\boldsymbol{w}$ is a *k-approximate cover* of $\boldsymbol{x}$ if $\boldsymbol{w}$ is a factor of $\boldsymbol{x}$ and every position of $\boldsymbol{x}$ lies within some k-approximate occurrence of $\boldsymbol{w}$ in $\boldsymbol{x}$. A border $\boldsymbol{u}$ of a string $\boldsymbol{y}$ is an *enhanced cover* of $\boldsymbol{y}$ if the number of positions of $\boldsymbol{y}$ which lie within some occurrence of $\boldsymbol{u}$ in $\boldsymbol{y}$ is the maximum among all borders of $\boldsymbol{y}$ [3].

A *deterministic finite automaton* $\mathcal{M}$ is a quintuple (Q, A, δ, q_0, F) where

- Q is a nonempty finite set of states,
- A is a nonempty finite input alphabet,
- $\delta : Q \times A \mapsto Q$ is a transition function (partially defined, i.e., for some pair (q, a), where $q \in Q, a \in A$, is $\delta(q, a)$ undefined),
- $q_0 \in Q$ is an initial state,
- $F \subseteq Q$ is a set of final states.

An *extended transition function* of a deterministic automaton $\mathcal{M} = (Q, A, \delta, q_0, F)$ is denoted by δ^* and it is defined for $q \in Q, a \in A, \boldsymbol{u} \in A^*$ inductively: $\delta^*(q, \varepsilon) = q$, $\delta^*(q, \boldsymbol{u}a) = \delta(\delta^*(q, \boldsymbol{u}), a)$. String $\boldsymbol{w}$ is *accepted* by $\mathcal{M}$ if and only if $\delta^*(q_0, \boldsymbol{w}) \in F$.

An automaton $\mathcal{M}$ *accepts a set of strings* B if and only if for all $\boldsymbol{u} \in B$ holds that $\boldsymbol{u}$ is accepted by $\mathcal{M}$. A *nondeterministic finite automaton* $\mathcal{M}_{\mathrm{N}}$ is a quintuple (Q, A, δ, q_0, F) where

- Q is a nonempty finite set of states,
- A is a nonempty finite input alphabet,
- $\delta : Q \times A \mapsto \mathcal{P}(Q)$ is a transition function,
- $q_0 \in Q$ is an initial state,
- $F \subseteq Q$ is a set of final states.

An *extended transition function* of a nondeterministic finite automaton $\mathcal{M}_{\mathrm{N}}$ is denoted by δ^* and it is defined for $q_1, q_2 \in Q, a \in A, \boldsymbol{u} \in A^*$ inductively: $\delta^*(q_1, \varepsilon) = \{q_1\}$, $\delta^*(q_1, \boldsymbol{u}a) = \bigcup_{q_2 \in \delta^*(q_1, \boldsymbol{u})} \delta(q_2, a)$. A *finite automaton* (also known as a finite state machine) is either a deterministic or a nondeterministic finite automaton. A *suffix automaton* for a string $\boldsymbol{u}$ is a finite automaton that accepts a set of all suffixes of $\boldsymbol{u}$. Let us have a set S of k-approximate suffixes of a string $\boldsymbol{u}$ defined as: $\boldsymbol{v} \in S$ if and only if for all suffixes $\boldsymbol{s}$ of $\boldsymbol{u}$, $\mathsf{H}(\boldsymbol{s}, \boldsymbol{v}) \leq k$; a *$k$-approximate suffix automaton* for the string $\boldsymbol{u}$ is a finite automaton that accepts a set of k-approximate suffixes S of $\boldsymbol{u}$. A *d-subset* of a state of a deterministic finite automaton is an ordered set of *elements*. Each element e is represented by two integers: $\mathsf{depth}(e)$ and $\mathsf{level}(e)$, where $\mathsf{depth}(e)$ corresponds to an end position and $\mathsf{level}(e)$ represents the Hamming distance of some factor of $\boldsymbol{u}$.

Definition 1 (k-approximate enhanced cover). *A string $\boldsymbol{w}$ is a k-approximate enhanced cover of a string $\boldsymbol{x}$ if $\boldsymbol{w}$ is a border of $\boldsymbol{x}$ and the number of positions of $\boldsymbol{x}$ which lie within some k-approximate occurrence of $\boldsymbol{w}$ in $\boldsymbol{x}$ is the maximum among all borders of $\boldsymbol{x}$.*

See an example of a k-approximate enhanced cover in Fig. 1.

Problem definition Given a string $\boldsymbol{w}$ and an integer k, the problem of *computing all k-approximate enhanced covers of $\boldsymbol{w}$* is to find all borders of $\boldsymbol{w}$ that satisfy Definition 1.

Related Work. The idea of a quasipediodic string (i.e., a string having a cover) was introduced by Apostolico and Ehrenfeucht [1], Moore and Smyth gave a linear-time algorithm for computing all covers of a given string [6,7]. An algorithm for computing all covers in generalized strings based on a suffix automaton was introduced by Voráček and Melichar [11].

Computing approximate covers was introduced by Sim et al. [9]. Christodoulakis et al. [2] implemented the algorithm based on dynamic programming and showed its practical time complexity for Hamming, edit and weighted edit distance. Guth, Melichar, and Balík [4] gave an algorithm for computing all approximate covers with the Hamming distance based on a suffix automaton.

In 2013, Flouri et al. [3] introduced a notion of the *enhanced cover* and gave a linear time algorithm for computing the minimum enhanced cover of a given string.

2 Problem Solution

2.1 Basic Idea

The presented solution of the problem of computing all k-approximate enhanced covers of a given string is based on the algorithm for computing all k-approximate

covers [4]. The referenced algorithm works in two phases: find candidate factors and compute the smallest Hamming distance for each candidate to cover the given string. To find the candidate factors, a subset construction [8][5, Alg. 1.40] of a deterministic k-approximate suffix automaton for the Hamming distance is used. This way, positions of each k-approximate occurrence of each factor of the given string is obtained. In the second phase, each factor is checked, whether it k-approximately covers the given string. To do that, subsequent positions (obtained by the subset construction) are compared with the factor length – there must be no gap between subsequent k-approximate occurrences of the factor. In order to reduce space complexity, only part of the deterministic automaton is stored in a memory – a depth-first search is done and all unnecessary states are removed.

The above mentioned algorithm [4] is used to solve the problem of computing all k-approximate enhanced covers after some modifications.

The idea found in [5, Section 4] and used in [4] is to use a nondeterministic k-approximate suffix automaton $\mathcal{M}_\mathrm{N}$ for a string $\boldsymbol{x}$ as an indexing structure. This automaton accepts all k-approximate suffixes of $\boldsymbol{x}$. Moreover, every string $\boldsymbol{u}$ "read" by $\mathcal{M}_\mathrm{N}$ reaches a set B of states, i.e. $\delta^*(q_0, \boldsymbol{u}) = B$. With the proper labelling, depth i of each such state $q \in B$ is equal to a k-approximate end position of $\boldsymbol{u}$ in $\boldsymbol{x}$, and level j of q is the minimum Hamming distance such that i is a j-approximate end position of $\boldsymbol{u}$ in $\boldsymbol{x}$ and there exists no $l < j$ such that i is an l-approximate end position of $\boldsymbol{u}$ in $\boldsymbol{x}$. Therefore, in addition to accept all k-approximate suffixes of $\boldsymbol{x}$, $\mathcal{M}_\mathrm{N}$ is able to identify all k-approximate end positions of all k-approximate factors of $\boldsymbol{x}$. See an example of a nondeterministic k-approximate suffix automaton in Fig. 2.

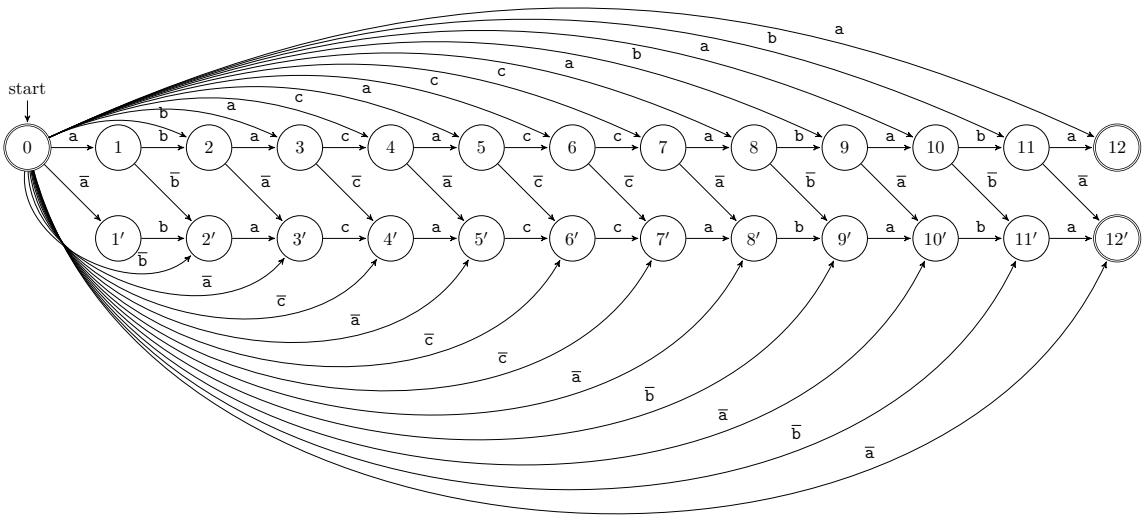

Figure 2. Example of a nondeterministic k-approximate suffix automaton for the string $\mathtt{abacaccababa} \in A^*$ ($\overline{\mathtt{a}}$ denotes supplement, i.e. $\overline{\mathtt{a}} = A \setminus \{\mathtt{a}\}$; a state depth is denoted by an integer, a state level is denoted by the number of primes, a final state is denoted by a double circle)

To obtain a k-approximate deterministic suffix automaton $\mathcal{M}$, the subset construction [5,8] may be used. Instead of the subset construction, similar algorithm

that represents states of $\mathcal{M}$ as subsets of $\mathcal{M}_{\mathrm{N}}$ with preserved depths and levels, is used in [4]. An advantage of the deterministic k-approximate suffix automaton for $\boldsymbol{x}$ over the nondeterministic one is that processing a string $\boldsymbol{u}$ using $\mathcal{M}$ takes linear time in the length of $\boldsymbol{u}$, regardless of the length of $\boldsymbol{x}$.

Note that in the algorithm presented in this paper, the nondeterministic automaton is not constructed, it is used just to describe the concept. Instead, states of the deterministic automaton are constructed directly, utilising the knowledge of the regular structure of $\mathcal{M}_{\mathrm{N}}$.

2.2 The Algorithm

From the definition of a k-approximate enhanced cover for a given string $\boldsymbol{x}$ follows that every k-approximate enhanced cover of $\boldsymbol{x}$ is a border (exact) of $\boldsymbol{x}$. Every border of $\boldsymbol{x}$ is accepted by a k-approximate suffix automaton for $\boldsymbol{x}$ and even by its part, a *backbone*.

Definition 2 (Backbone). *[5, Def. 3.12 and Sec. 3.4.1] Assume a k-approximate deterministic suffix automaton $\mathcal{M} = (Q, A, \delta, q_0, F)$ for a string $\boldsymbol{x}$. A* backbone *of $\mathcal{M}$ is a deterministic automaton $\mathcal{M}_B = (Q_B, A, \delta_B, q_0, F_B)$ such that for all $0 < i \leq |\boldsymbol{x}|$ holds $Q_B = \{q_i : q_i \in Q\}$, $\delta_B(q_{i-1}, \boldsymbol{x}[i]) = q_i$, and $F_B = \{q : q \in Q_B \cap F\}$.*

In other words, the backbone is the part of $\mathcal{M}$ that enables "reading" of $\boldsymbol{x}$ (and of all its prefixes) exactly. See an example of a backbone in Fig. 3.

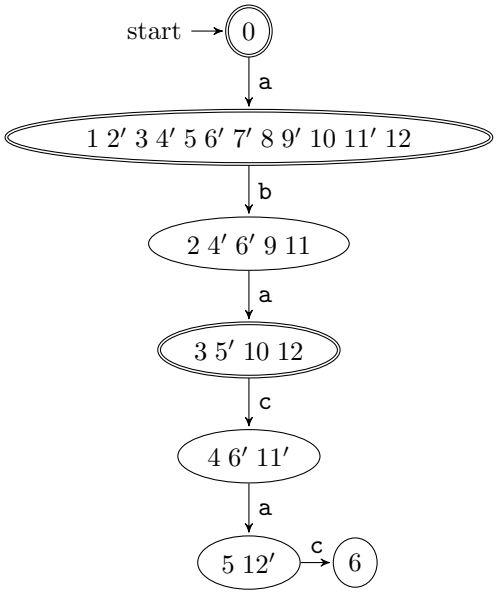

Figure 3. An example of a backbone of a deterministic k-approximate suffix automaton for the string `abacaccababa` – the part useful for computing borders (the d-subset element depth is denoted by an integer, the level is denoted by number of primes, a final state is denoted by a double circle)

In order to find k-approximate enhanced covers of $\boldsymbol{x}$ among its borders, the number of symbols of $\boldsymbol{x}$ that lie within some k-approximate occurrence of the border must be computed. This may be obtained from k-approximate occurrences of the border by summing the letters that lie within each occurrence, counting each letter only once. Considering each two subsequent k-approximate positions $i, j; i < j$ of a border $\boldsymbol{p}$ of $\boldsymbol{x}$, there are three cases:

```
   Input  : A state q of a deterministic k-approximate suffix automaton for x
   Output: The number of letters of x covered by a border corresponding to q
 1 begin
 2 |   r ← 0;
 3 |   e_f is the first d-subset element of q (one with the minimum depth);
 4 |   m ← depth(e_f);
 5 |   E is an array of d-subset elements of q having the same order as they are appended;
 6 |   for i ∈ 2..|E| do
 7 |   |   if depth(E[i]) − depth(E[i − 1]) < m then
 8 |   |   |   r ← r + depth(E[i]) − depth(E[i − 1]);
 9 |   |   else
10 |   |   |   r ← r + m;
11 |   |   end
12 |   end
13 |   return r;
14 end
```

Function `distEnhCov`

overlap $(j - i < |\boldsymbol{p}|)$ add $j - i$ to the number of letters covered by $\boldsymbol{p}$
square $(j - i = |\boldsymbol{p}|)$ add $|\boldsymbol{p}|$ to the number of letters covered by $\boldsymbol{p}$
gap $(j - i > |\boldsymbol{p}|)$ add $|\boldsymbol{p}|$ to the number of letters covered by $\boldsymbol{p}$

The pseudocode of this computation is listed as Function `distEnhCov`. All the k-approximate occurrences of each border of $\boldsymbol{x}$ are obtained from d-subsets of the backbone of the deterministic k-approximate suffix automaton for $\boldsymbol{x}$.

As in the algorithm for computing all k-approximate covers [4], the number of covered symbols of $\boldsymbol{x}$ for each of its borders is computed just after the state of the backbone is constructed. This state may be removed just after the next state is constructed, therefore at most two states are needed to be stored at a time. Unlike in the algorithm in [4], all the borders with the maximum number of covered symbols must be stored along with their number of covered symbols, because it is unknown what the number is, before the algorithm finishes.

In order to further reduce space complexity, the borders with the maximum number of covered symbols are not actually stored directly. Because every k-approximate enhanced cover is a prefix, the length is enough to specify it and therefore only the prefix length is stored and reported (variable p in Algorithm 1).

Example 3 (Computing all k-approximate enhanced covers). Let us have a string $\boldsymbol{x} =$ `abacaccababa` and maximum Hamming distance $k = 1$. The set of all 1-approximate enhanced covers of $\boldsymbol{x}$ is computed using Alg. 1. A d-subset of the first state q_1 of the backbone (see Fig. 3) is $1\ 2'\ 3\ 4'\ 5\ 6'\ 7'\ 8\ 9'\ 10\ 11'\ 12$. Because the related prefix length is $p = 1$ and $k = 1$, no meaningful result may be obtained for this state. A d-subset of the second constructed state is $2\ 4'\ 6'\ 9\ 11$. After its construction, q_1 and its d-subset are removed from a memory. Because depth of the last d-subset element is 11, it is not a final state (and does not represent a border of $\boldsymbol{x}$). A d-subset of the next constructed state is $3\ 5'\ 10\ 12$. Again, the previous state is now removed. Because the depth of the last element is 12 (equal to the length of $\boldsymbol{x}$) and its level is 0, the related prefix `aba` is a border of $\boldsymbol{x}$. The number of positions of $\boldsymbol{x}$ covered by 1-approximate occurrences of `aba` in $\boldsymbol{x}$ is now computed. The end positions (read from the d-subset) are 3, 5, 10, 12 and therefore the number of covered positions is 10. This is the maximum, the variable h is updated and `aba` is added to the set C. The subsequent backbone state is not final and the next state $5\ 12'$ is not final as well

```
Input  : A string x, the maximum Hamming distance k
Output: A set C of k-approximate enhanced covers of x (border lengths)
1  begin
2      C ← ∅;
3      h ← 0;
4      q_1 is a state;
5      for i ∈ 1..|x|;                          // construct a d-subset of the first state
6      do
7          e is a d-subset element such that depth(e) ← i;
8          if x[1] = x[i] then
9              level(e) ← 0;
10             append e to q_1
11         else if k > 0 then
12             level(e) ← 1;
13             append e to q_1
14         end
15     end
16     p ← 1;                                                    // a prefix length
17     q_p ← q_1;
18     for i ∈ 2..|x| do
19         p ← p + 1;
20         q_n is a state;
21         for e_p ∈ q_p;                       // construct a d-subset of the next state
22         do
23             if depth(e_p) < |x| then
24                 e_n is a d-subset element such that depth(e_n) ← depth(e_p) + 1;
25                 if x[i] = x[depth(e_n)] then
26                     level(e_n) ← level(e_p);
27                     append e_n to q_n;
28                 else if level(e_p) < k then
29                     level(e_n) ← level(e_p) + 1;
30                     append e_n to q_n;
31                 end
32             end
33         end
34         destroy q_p;
35         if number of d-subset elements of q_n is less than 2 then
36             stop ;                                  // all borders of x are examined
37         end
38         e_l is the last d-subset element of q_n;
39         if depth(e_l) = |x| and level(e_l) = 0 and |p| > k;                  // q_n is final
40         then
41             h_n ← distEnhCov (q_n);
42             if h_n > h then
43                 h ← h_n;
44                 C ← ∅;
45             end
46             if h_n = h then
47                 append p to C;
48             end
49         end
50     end
51 end
```

Algorithm 1: Computing all k-approximate enhanced covers

(although the last element depth is 12, its level is 1, i.e. it represents an approximate occurrence and therefore not an exact border of $\boldsymbol{x}$). The subsequent state d-subset contains only one element and therefore cannot represent a border (it represents a string that occurs only once in $\boldsymbol{x}$). It is not needed to construct any further state, as the construction starting at line 21 cannot create a state with multiple d-subset elements.

2.3 Time and Space Complexity

Theorem 4 (Space complexity). *Given a string $\boldsymbol{x}$, $|\boldsymbol{x}| = n$, and an integer k, Algorithm 1 needs at most $4n$ space.*

Proof. The following is needed to be stored in a memory: the input string $\boldsymbol{x}$, the set of results C, a length p of actual prefix, states and d-subsets. A state is represented by its d-subset. Every d-subset is an array of elements and its size is always known. Each d-subset element is represented by two integers, every integer used in this algorithm is between 0 and $\max(n, k)$.

The for loop starting at line 6 is iterated n times. Within each iteration, at most one d-subset element is added, therefore q_1 needs at most $2n + 1$ space.

During construction of a next state q_n (starting at line 21), for each d-subset element of q_p (there are at most n), a new d-subset element is constructed. The new element is not stored in two cases only: exceeding the maximum level k, or the depth of the element over the length of $\boldsymbol{x}$ (checked at line 23). Therefore the maximum number of d-subset elements of a state are: n for a first state, $n - 1$ for a second state, etc. Due to the line 34, at most two states are in a memory at a time, therefore d-subset elements need at most $4n$ space. For some states, the integer p is added to C, so total maximum size of C is n. □

Theorem 5 (Time complexity). *Given a string $\boldsymbol{x}$, $|\boldsymbol{x}| = n$, and an integer k, Alg. 1 needs at most $\mathcal{O}(n^2)$ time.*

Proof. The for loop starting at the line 6 needs $\mathcal{O}(n)$ time. The for loop starting at the line 21 needs $\mathcal{O}(|q_p|)$ time (see the proof of Theorem 4). All the statements whithin this loop are evaluated in constant time (a d-subset is an array, each element is an object with two associated integers). Therefore the evaluation of the for loop (the line 21) takes $\sum_{i=2}^{n} n - i + 1 = \frac{n^2-n}{2}$. The line 34 takes the same time as the above for loop. Computing the Function `distEnhCov` takes at most $5(n - 2)$ for the second state (recall the d-subset size in the proof of Theorem 4), so it takes $\frac{5n^2-15n}{2}$ for the whole input. □

Example 6. For the input string $\boldsymbol{x} = \mathtt{aa}\cdots\mathtt{a}$, quadratic time regarding $|\boldsymbol{x}|$ is needed for Alg. 1.

3 Experimental Results

The algorithm has been implemented using the C++ programming language. It has been compiled using the gcc 5.3.0 with the O3 optimisation level, and run on the i5-2520M (4-core) machine under the Hardened Gentoo Linux 4.3.5 with disabled swap. As input data, Saccharomyces cerevisiae S288c chromosome IV[1] was used. For various

[1] The sequence was downloaded from `http://www.ncbi.nlm.nih.gov/nuccore/NC_001136.10`.

input lengths, the input string consists the of first n characters of the chromosome. For each input length n and the maximum Hamming distance k, the following values were measured using the GNU time utility:

- elapsed time as total number of CPU-seconds that the implemented program spent in user mode,
- memory consumption as maximum resident set size of the implemented program.

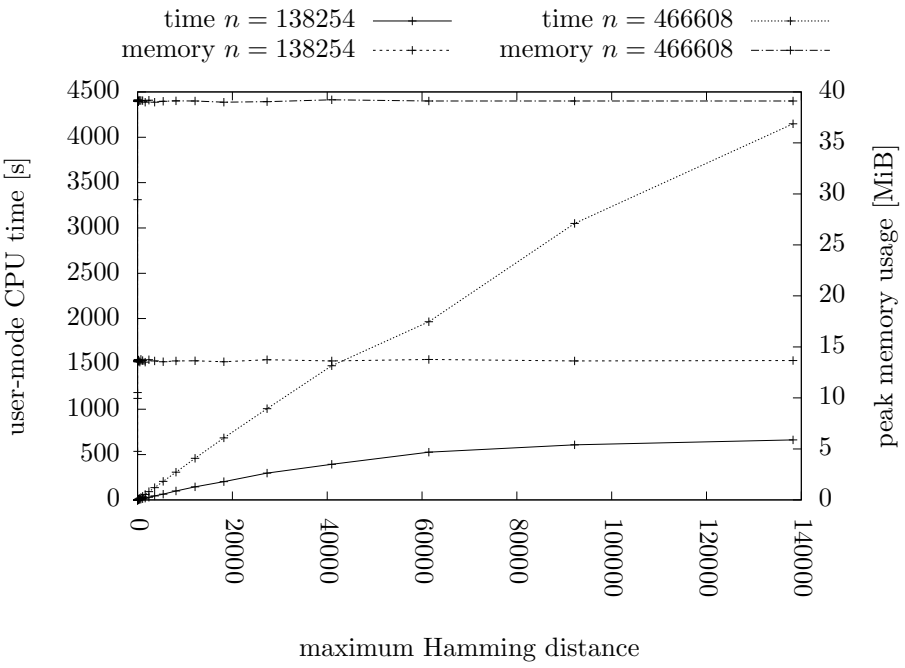

Figure 4. Time and memory consumption, when processing the chromosome, depending on the maximum distance k when the input length n is fixed

Both the time and memory consumption is shown in Figs. 4 and 5. Values of the time consumption are always shown in seconds at the left border of each figure, values of memory consumption are shown in megabytes at the right border of each figure. It is distinguished by a line type whether the time or the memory consumption is being plotted. Because the consumption depends on both the input length and the maximum allowed distance, the distance is fixed in the plots shown in Fig. 5 and the input length is fixed in the plot shown in Fig. 4. The value of the fixed length, or the distance, respectively, is shown in a key of each figure, and distinguished by a line type.

In the plots shown in Fig. 5, the time and memory consumption is shown depending on varying input string length when the maximum allowed Hamming distance is fixed to a few arbitrary values.

Note that although the time and space complexity (Theorems 4 and 5) do not depend on k, the real consumption is varying for different k. The reason is that the complexities in the above theorems are maximal, however, for the data used in the experiment, k is limiting the number of d-subset elements. The limiting effect of k is better shown in Fig. 4. In the plots shown in this figure, the consumption is shown depending on varying maximum allowed Hamming distance while the input string length is fixed to a few arbitrary values.

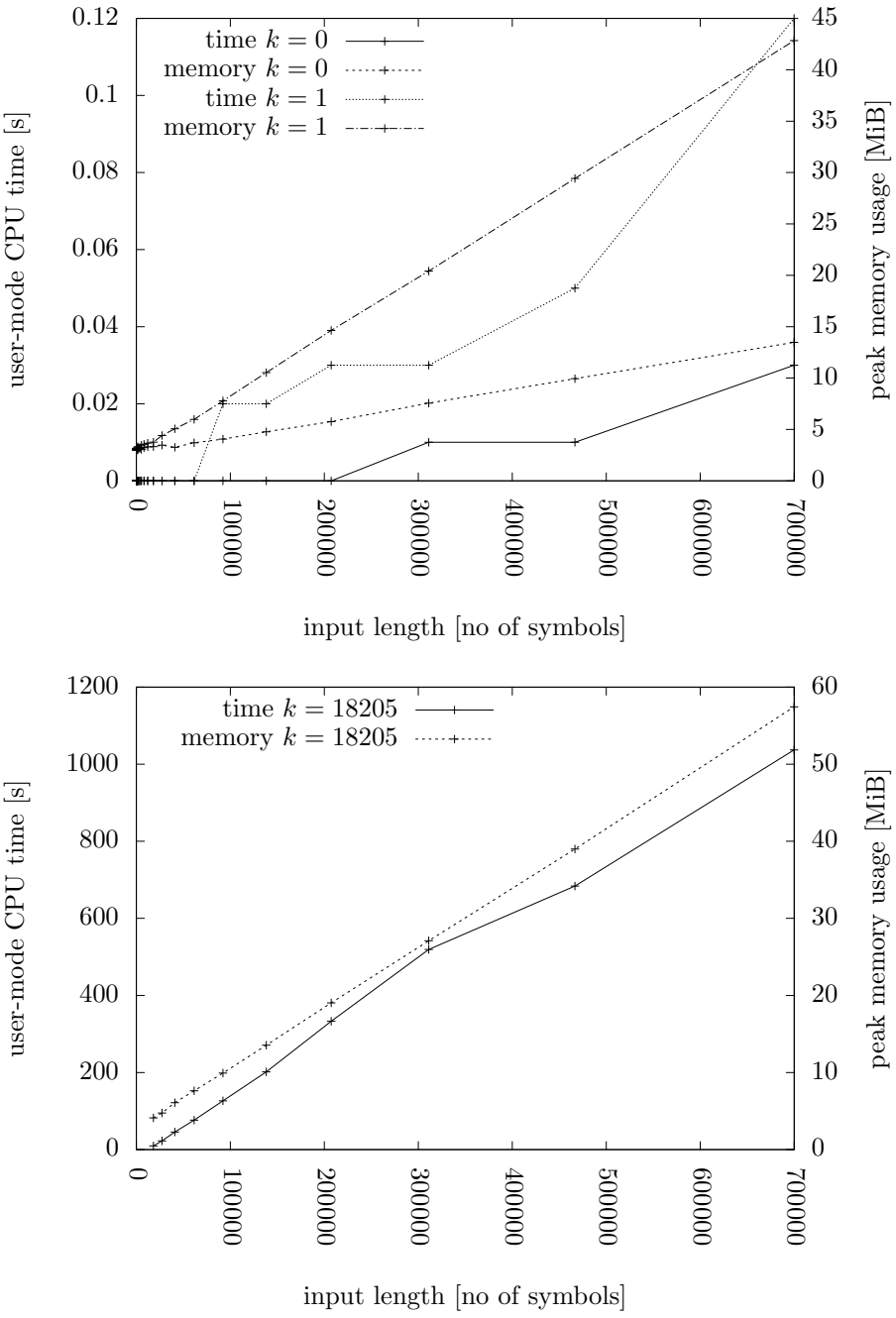

Figure 5. Time and memory consumption, when processing the chromosome, depending the on string size when the maximum Hamming distance k is fixed

The plots in Fig. 5 seem to show linear growth of the time consumption depending on the input string length. This is due to the input data (small number of repeating factors). With a regular input string, e.g. `ab` repeating many times, time needed for processing the string is apparently quadratic (see Fig. 6) with respect to the input string length (according to Theorem 5).

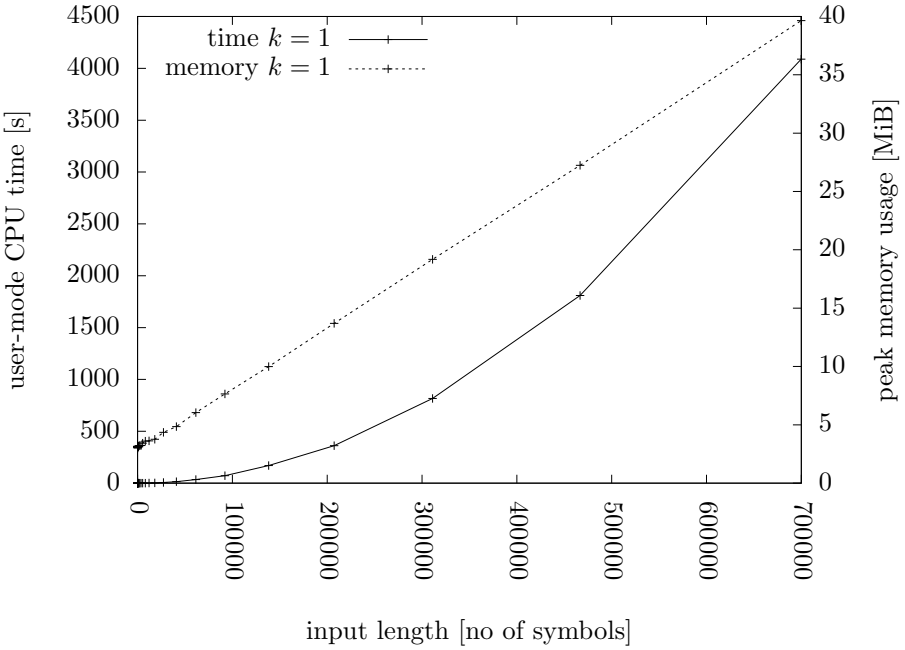

Figure 6. The time and memory consumption for input string $(\texttt{ab})^{4026308608}$ depending on the input string length when the maximum Hamming distance $k = 1$ is fixed

4 Conclusions

In this paper, new problem related to string covering has been stated and an algorithm solving the problem has been presented.

As future work, problem solutions with less than quadratic time complexity may be explored. It is probably achievable using a different data structure and technique than indexing with the subset construction of a suffix automaton. Also, the problem statement may be extended to find approximate borders that cover the maximum number of positions of a given string among all approximate borders. Also the notion of an enhanced left-cover array, introduced in [3], may be extended to accommodate the Hamming distance.

References

1. A. Apostolico and A. Ehrenfeucht: *Efficient detection of quasiperiodicities in strings.* Theoretical Computer Science, 119(2) 1993, pp. 247–265.
2. M. Christodoulakis, C. S. Iliopoulos, K. S. Park, and J. S. Sim: *Implementing approximate regularities.* Mathematical and Computer Modelling, 42 October 2005, pp. 855–866.
3. T. Flouri, C. S. Iliopoulos, T. Kociumaka, S. P. Pissis, S. J. Puglisi, W. Smyth, and W. Tyczyński: *Enhanced string covering.* Theoretical Computer Science, 506 2013, pp. 102–114.
4. O. Guth, B. Melichar, and M. Balik: *Searching all approximate covers and their distance using finite automata.* Information technologies–applications and theory, 2008, pp. 21–26.
5. B. Melichar, J. Holub, and T. Polcar: *Text searching algorithms, Volume I*, November 2005, Available from: `http://stringology.org/athens`.
6. D. Moore and W. F. Smyth: *An optimal algorithm to compute all the covers of a string.* Information Processing Letters, 50 1994, pp. 101–103.
7. D. Moore and W. F. Smyth: *A correction to "An optimal algorithm to compute all the covers of a string".* Information Processing Letters, 54(2) 1995, pp. 101–103.
8. M. O. Rabin and D. Scott: *Finite automata and their decision problems.* IBM journal of research and development, 3(2) 1959, pp. 114–125.
9. J. S. Sim, K. S. Park, S. R. Kim, and J. S. Lee: *Finding approximate covers of strings.* Journal of Korea Information Science Society, 29 2002, pp. 16–21.
10. W. Smyth: *Computing regularities in strings: a survey.* European Journal of Combinatorics, 34(1) 2013, pp. 3–14.
11. M. Voráček and B. Melichar: *Searching for regularities in strings using finite automata*, in Proceedings of Workshop 2005, vol. A, Czech Technical University in Prague, 2005, pp. 264–265.