**Algorithms for Approximate K-Covering of Strings**

By: Lili Zhang and F. Blanchet-Sadri


***Reprinted with permission. No further reproduction is authorized without written permission from World Scientific Publishing. This version of the document is not the version of record. Figures and/or pictures may be missing from this format of the document.***

Made available courtesy of World Scientific Publishing: [http://dx.doi.org/10.1142/S0129054105003789](http://dx.doi.org/10.1142/S0129054105003789)

**Abstract:** Computing approximate patterns in strings or sequences has important applications in DNA sequence analysis, data compression, musical text analysis, and so on. In this paper, we introduce approximate k-covers and study them under various commonly used distance measures. We propose the following problem: "Given a string x of length n, a set U of m strings of length k, and a distance measure, compute the minimum number t such that U is a set of approximate k-covers for x with distance t". To solve this problem, we present three algorithms with time complexity $O(km(n-k)), O(mn^2)$ and $O(mn^2)$ under Hamming, Levenshtein and edit distance, respectively. A World Wide Web server interface has been established at [http://www.uncg.edu/mat/kcover/](http://www.uncg.edu/mat/kcover/) for automated use of the programs.

**Keywords:** Strings; k-Covers; Approximate k-covers; Distance measures; String algorithms; Dynamic programming.

**Article:**

1. **Introduction**

A string v is called a cover of a string x if x can be constructed by concatenating or overlapping copies of v, so that every position of x lies within an occurrence of v. For example, TCAT is a cover of TCATTTCATCAT. This notion was introduced by Apostolico et al. in [3]. There, the shortest cover problem or the problem of computing the shortest cover of a given string x of length n was considered and an $O(n)$ time algorithm was described for this problem. Other linear time algorithms followed that improve on their result: In [4], Breslauer gives an on-line algorithm for the shortest cover problem thus computing the shortest cover of every prefix of x; In [10, 11], Moore and Smyth give an algorithm for the all covers problem or the problem of computing all the covers of x; Finally, in [9], Li and Smyth extend this result considerably by computing on-line all the covers of every prefix of x. PRAM (parallel random access machine) algorithms have also been developed for the shortest cover [5] and all covers [6] problems. Iliopoulos and Park gave an optimal $O(\log \log n)$ time algorithm for the shortest cover and all covers problems [6]. Apostolico and Ehrenfeucht considered yet another problem related to covers [2].

Given a string x, a set V of strings is called a set of covers for x (or V covers x) if x can be constructed by concatenating or overlapping strings in V. For example, the set {CTA, CTAC} covers CTACCTACTA. In addition, if each string in V has length k, then V is a set of k-covers for x. In [7], Iliopoulos and Smyth give an $O(n^2(n-k))$ time on-line algorithm for computing a minimum set of k-covers for a given string of length n.

A natural extension of the above problems is to allow errors when computing patterns. In some applications, specifically DNA sequence analysis, it becomes necessary to recognize u as an occurrence of v if the difference or distance between u and v is bounded by a certain threshold. Several definitions of distance have been proposed like the Hamming, Levenshtein and edit distances. In [1], Agius et al. give polynomial time algorithms to solve problems related to approximate covers according to these and other definitions of distance extending previous work by Sim et al. [15] (other results on approximate patterns in strings appear in [8, 13]).
In this paper, we introduce the notion of a set of approximate \textit{k}-covers. To our knowledge, no results are known about these approximate patterns. In Section 2, as a foundation for approximate \textit{k}-covering, we discuss Iliopoulos and Smyth's algorithm for \textit{k}-covering. In Section 3, we suggest the following problem: "Given a string \(x\), a set \(U\) of strings of length \(k\), and a distance measure, compute the minimum number \(t\) such that \(U\) is a set of approximate \textit{k}-covers for \(x\) with distance \(t\)." In Sections 4, 5 and 6, we give polynomial time algorithms to solve this problem under Hamming, Levenshtein and edit distance, respectively.

First, we review some basic concepts on strings. Let \(\Sigma\) be a nonempty finite set, or an \textit{alphabet}. A string (or \textit{word}) \(x\) over \(\Sigma\) is a finite concatenation of characters from \(\Sigma\). The \textit{length} of \(x\), or the number of characters in \(x\), is denoted by \(|x|\). A string of length \(n\) is sometimes called an \textit{n-string}. For any string \(x\) and \(i \leq j\), \(x[i..j]\) is the \textit{substring} of \(x\) of length \(j - i + 1\) that starts at position \(i\) and ends at position \(j\) (\(x\) is called a \textit{superstring} of \(x[i..j]\)). In particular, \(x[1..j]\) is the \textit{prefix} of \(x\) that ends at position \(j\) and is the \textit{suffix} of \(x\) that begins at position \(i\). The substring \(x[i..j]\) is the \textit{empty string} if \(i > j\) (the empty string is denoted by \(\emptyset\)). For example, \(\text{ACAAACC}\) is a string over the alphabet \{A, C\}, \(\text{CAA}\) is a substring, \(\text{ACAA}\) is a prefix, and \(\text{CC}\) is a suffix. The set of all strings over \(\Sigma\) is denoted by \(\Sigma^*\), and the cardinality of a subset \(X\) of \(\Sigma^*\) by \(|X|\).

2. Algorithm for \textit{k}-Covering

In this section, we present Iliopoulos and Smyth's \(O(n^2(n - k))\) time on-line algorithm for computing a minimum set of \textit{k}-covers for all prefixes of a given string \(x\) of length \(n\) [7]. Here we provide details on how to compute the cardinality of a minimum set of \textit{k}-covers for \(x\), and how to compute at least one such set. Lemma 1 below gives the reason for not computing all the minimum sets (there may be an exponential number of them).

First, we define the notion of a minimum set of \textit{k}-covers.

\textbf{Definition 1 ([7])} Given a string \(x\) and a positive integer \(k\) satisfying \(k < jx\), a set \(V\) of \(k\)-strings is called a set of \textit{k}-covers for \(x\) if \(V\) covers \(x\). Moreover, \(V\) is called minimum if \(|V|\) is a minimum.

For example, both \{\text{ACA, CAG, GTT}\} and \{\text{ACA, GTT}\} are sets of \(3\)-covers for \text{ACACAGTT} with the latter one being a minimum set.

The following are some basic facts about the minimum sets of \textit{k}-covers for a string \(x\) of length \(n\):

\textbf{Fact 1([7])} The strings \(x[1..k]\) and \(x[n - k + 1..n]\) are both elements of every minimum set of \textit{k}-covers for \(x\).

\textbf{Fact 2([7])} The cardinality of a minimum set of \textit{k}-covers for \(x\) is at most \([n/k]\). Indeed, the set

\[
\{x[i..i + k] \mid i = 0,1, \ldots, [n/k] - 1\} \cup \{x[n - k + 1..n]\}
\]

covers \(x\).

\textbf{Fact 3([7])} A minimum set of \textit{k}-covers for \(x\) is not necessarily unique. (For example, both \{\text{AAC, ACC, TTG}\} and \{\text{AAC, CCT, TTG}\} are minimum sets of \(3\)-covers for \text{AACCTTG}.)

It follows from the next lemma that the number of minimum sets of \textit{k}-covers for a string of length \(n\) may be exponential in \(n\).

\textbf{Lemma 1 ([7])} Let \(x\) be a string of length \(n\) whose symbols are all distinct, that is, for every pair of positions \(i, i'\) in \(x\), \(x[i] = x[i']\) if and only if \(i = i'\). Put \(n = hk - j\) where \(h, j\) are integers satisfying \(h > 2\) and \(0 < j < k\). If \(N_{j,h}\) denotes the number of distinct minimum sets of \textit{k}-covers for \(x\), then

\[N_{j,h} = \sum_{0 \leq j \leq h} N_{t,h-1}\] for every \(h \geq 3\), and
(b) \( N_{j,h} \in \Theta((j+1)^{h-1}) \).

We now outline our version of Iliopoulos and Smyth's algorithm which works iteratively computing the cardinalities of minimum sets of \( k \)-covers for all prefixes of a given string \( x \). Initially, the algorithm uses the idea from Fact 1 in order to compute the cardinalities of minimum sets of \( k \)-covers for the prefixes \( x[1..k+1] \), \( x[1..k+2] \), \( x[1..2k] \) of \( x \). For \( k < i \leq 2k \), if \( x[1..k] = x[i-k+1..i] \) then the minimum set of \( k \)-covers for \( x[1..i] \) is \( \{x[1..k]\} \) and the cardinality is 1; otherwise, the minimum set of \( k \)-covers for \( x[1..i] \) is \( \{x[1..k], x[i-k+1..i]\} \) and the cardinality is 2. For \( i > 2k \), the algorithm uses the idea that every minimum set of \( k \)-covers for \( x[1..i+1] \) depends only on the minimum sets computed for the previous \( k \) positions, that is, the minimum sets of \( k \)-covers for \( x[1..i], x[1..i-1], \ldots, x[1..i-k+1] \).

The following lemmas provide the other main ideas for the algorithm.

**Lemma 2 ([7])** For \( i \geq 2k \), let \( V_{i,1}, V_{i,2}, \ldots \) be the distinct minimum sets of \( k \)-covers for \( x[1..i] \). Put \( c_i = ||V_{i,1}|| = ||V_{i,2}|| = \ldots \) Then

\[
c_{i+1} = \min_{i-k<j \leq i} h ||V_{j,h} \cup \{x[i-k+2..i+0]\}||.
\]

**Lemma 3 ([7])** For \( i > 2k \), every minimum set \( V_{i+1,h} \) is a superset of some minimum set \( V_{j,h} \), with \( i-k < j \leq i \). Indeed, there exist \( i - k < j \leq i \) and \( h' \) such that

\[
V_{i+1,h} = V_{j,h} \cup \{x[i-k+2..i+1]\}.
\]

**Lemma 4 ([7])** For \( i \geq 2k \), suppose that \( V_{i+1,h} \supseteq V_{i,h'} \) for some \( i-k \leq j \leq i \) and some \( h' \). Then \( c_{i+1} = c_j \) if \( x[i-k+2..i+1] \in V_{j,h'} \); \( c_{i+1} = c_j + 1 \) otherwise.

As observed before, for \( i > 2k \), there exist \( i-k < j \leq i \) and \( h' \) such that \( V_{i+1,h} = V_{i,h'} \cup \{x[i-k+2..i+1]\} \). This could be the basis for an algorithm to compute all the minimum sets of \( k \)-covers for \( x[1..i+1] \). However, by Lemma 1, the number of such minimum sets for any value of \( j \) may be exponential in \( j \), leading to an inefficient algorithm. To achieve efficiency, the following data structures are used:

- An integer array \( c \)
  \( c[i] \), where \( k < i \leq n \), records the cardinality of every minimum set of \( k \)-covers for \( x[1..i] \).
- A 2-dimensional Boolean array \( A \)
  \( A[i,j] \), where \( k < i \leq n \) and \( k \leq j \leq i \), records TRUE if the \( k \)-string \( x[j-k+1..j] \) is an element of at least one of the minimum sets for \( x[1..i] \); \( A[i,j] \) records FALSE otherwise.
- A global integer array \( L \)
  \( L[i] \), where \( k \leq i \leq n \), records the minimum integer \( j \) distinct from \( i \) such that \( x[i-k+1..i] = x[j-k+1..j] \) if such \( j \) exists; \( L[i] \) records \( i \) otherwise.
- A Boolean array \( MARK \)
  \( MARK[i'] \), where \( k \leq i-k < i' \leq i < n \), records TRUE if there exists \( j' \) such that \( A[i',j'] = TRUE \) and \( x[j'-k+1..j'] = x[i-k+2..i+1] \); \( MARK[i'] \) records FALSE otherwise.

**Algorithm** \( k \)-Covering

The algorithm consists of three steps.

**Step 1:** For \( k < i \leq 2k \), initialize \( c[i] \) with 1 if \( x[i-k+1..i].x[1..k] \), and with 2 otherwise. For \( k < i \leq 2k \) and \( k \leq j \leq i \), initialize \( A[i,j] \) with TRUE if \( j = k \) or \( j = i \), and with FALSE otherwise.
Step 2: For $k \leq i \leq n$, compute the minimum integer $j$ such that $k \leq j \leq n$, $j \neq i$, and $x[i - k + 1..i] = x[j - k + 1..j]$. If such $j$ is found, set $L[i] = j$; otherwise, set $L[i] = i$.

Step 3: For $2k \leq i < n$, compute $c[i + 1]$ and $A[i + 1, -]$.

- For $i - k < j \leq i$, use array $L$ (from Step 2) to compute $\text{MARK}[j]$. If $L[i + 1] \leq j$, then $\text{MARK}[j] = \text{TRUE}$; otherwise, $\text{MARK}[j] = \text{FALSE}$. In the process, compute $c[i + 1]$ according to the formula:

$$c[i + 1] = \min_{i - k < j \leq i} (c[j] \text{ if } \text{MARK}[j] = \text{TRUE}, c[j] + 1 \text{ otherwise})$$

- Using Fact 1, set $A[i + 1, i + 1] = \text{TRUE}$. Now, there exists at least one value of $j$, $i - k < j \leq i$, satisfying Eq. (1). Denote such $j$ by $i'$. For $k \leq j' \leq i$, if $A[i', j'] \text{ TRUE}$, then set $A[i + 1, j'] = \text{TRUE}$; otherwise, set $A[i + 1, j'] = \text{FALSE}$.

When all computations are done, Algorithm $k$-Covering returns $c$.

Note: For $k < i \leq n$, in order to compute a minimum set of $k$-covers for $x[1..i]$, pick up $c[i]$ entries in row $i$ of $A$ that are $\text{TRUE}$: say, $A[i, j_1], \ldots, A[i, j_{c[i]}]$ where $k \leq j_1 < \cdots < j_{c[i]} < i$. If the set

$$V_i = \{x[j_1 - k + 1..j_1], \ldots, x[j_{c[i]} - k + 1..j_{c[i]}]\}$$

is of cardinality $c[i]$ and covers $x$, then 14 is as desired.

We now express the algorithm in pseudo programming language code.

Algorithm $k$-Covering

**input:** string $x$ of length $n$ and positive integer $k \leq n$

**output:** cardinality of a minimum set of $k$-covers (as well as a minimum set of $k$-covers) for every prefix of $x$

// Step 1: Initialize $c$ and $A$

```plaintext
for $l \leftarrow k + 1$ to $2k$ do
    if $x[i - k + 1..i] = x[1..k]$ then $c[i] \leftarrow 1$
    else $c[i] \leftarrow 2$
    for $j \leftarrow i$ do
        if $j = k$ or $j = i$ then $A[i, j] \leftarrow \text{TRUE}$
        else $A[i, j] \leftarrow \text{FALSE}$
```

// Step 2: Compute $L$

```plaintext
for $l \leftarrow k$ to $n$ do
    $L[i] \leftarrow i$
    flag $\leftarrow 0$
    for $j \leftarrow k$ to $n$ do
        if flag $0$ and $j \neq i$ and $x[i - k + 1..i] = x[j - k + 1..j]$ then
            $L[i] \leftarrow j$
            flag $\leftarrow 1$
```

// Step 3: Compute $c$ and $A$
for $i \leftarrow 2k$ to $n - 1$ do
  $c[i + 1] \leftarrow \infty$
  for $j \leftarrow i - k + 1$ to $i$ do
    if $L[i + 1] \leq j$ then $\text{MARK}[j] \leftarrow \text{TRUE}$
    if $c[i + 1] > c[j]$ then $c[i + 1] \leftarrow c[j]$
    else $\text{MARK}[j] \leftarrow \text{FALSE}$
    if $c[i + 1] > c[j] + 1$ then $c[i + 1] \leftarrow c[j] + 1$
  $A[i + 1, i + 1] \leftarrow \text{TRUE}$
  for $j' \leftarrow k$ to $i$ do
    if $(\text{MARK}[i'] = \text{TRUE} \text{ and } c[i + 1] = c[i'])$ or
      $(\text{MARK}[i'] = \text{FALSE} \text{ and } c[i + 1] = c[i'] + 1)$ then
      if $A[i', j'] = \text{TRUE}$ then $A[i + 1, j'] \leftarrow \text{TRUE}$
    else $A[i + 1, j'] \leftarrow \text{FALSE}$
return $c$

**Theorem 1** Algorithm $k$-Covering computes in $O(k(n - k)^2)$ time a minimum set of $k$-covers for every prefix of a given string of length $n$.

We now illustrate the algorithm with the following example.

**Example 1** Given the string $x = \text{TCATCATCTCAT}$ of length 12 and the positive integer $k = 4$, Algorithm $k$-Covering computes the cardinality of minimum sets of 4-covers for $x$ as $c[12] = 2$, and computes such a minimum set of 4-covers as $\{\text{TCAT}, \text{CATC}\}$ for instance.

**3. Approximate k-Covering**

In some applications, it becomes necessary to recognize the string $u$ as an occurrence of the string $v$ if the distance between $u$ and $v$ is bounded by a certain threshold. There are several well-known distance measures which focus on transforming $u$ into $v$ by a series of operations on individual characters, each operation having cost 1. The distance $\delta(u, v)$ between $u$ and $v$ is then the minimum cost to transform $u$ into $v$. For the Levenshtein distance, the allowed operations are insertion of a character into $u$, the deletion of a character from $u$, or the substitution of a character in $u$ with a character in $v$; For the Hamming distance, insertions and deletions are not allowed; And for the edit distance, substitutions are not allowed. It also becomes necessary to relax the conditions of a set $V$ of $k$-covers for a given string $x$ and to recognize $U$ as an occurrence of $V$ if $U$ is a set of approximate $k$-covers for $x$ with distance $t$. We state this idea more precisely in the following definition.

**Definition 2** Let $t$ be a nonnegative integer and $\delta$ be a distance measure. Given a string $x$ and a positive integer $k$ satisfying $k \leq |x|$ a set $U$ of $k$-strings is called a set of approximate $k$-covers for $x$ with distance $t$ if there exists a (multi)set $V$ such that the following conditions hold:

- The (multi)set $V$ corresponds to a sequence of substrings of $x$, $v_1, v_2, ..., v_t$ such that $v_1$ starts at position $i_1$ of $x$, $v_2$ starts at position $i_2$ of $x$, ..., with $1 \leq i_1 \leq i_2 \leq \cdots$ and with $V$ covering $x$.
- For every $u \in U$, there exists $v \in V$ such that $\delta(u, v) \leq t$.
- For every $v \in V$, there exists $u \in U$ such that $\delta(u, v) \leq t$.

The set $V$ is said to be generated by $U$. Moreover, if $u \in U$, $v \in V$ and $\delta(u, v) \leq t$, then $v$ is said to be generated by $u$ or $u$ is called a generator for $v$.

In the next three sections we consider the following problem under Hamming, Levenshtein and edit distances: "Given a string $x$ of length $n$, a set $U$ of $m$ strings of length $k$, and a distance measure, compute the minimum number $t$ such that $U$ is a set of approximate $k$-covers for $x$ with distance $t$". We classify our problem into three versions: the Hamming distance version (Problem $t_h$ and $O(km(n - k))$ time Algorithm $t_h$ described in Section 4), the Levenshtein distance version (Problem $t_l$ and $O(mn^2)$ time Algorithm $t_l$ described in Section 5), and the edit
distance version (Problem \( t_e \) and \( O(mn^2) \) time Algorithm \( t_e \) described in Section 6). For a preview, we illustrate the different outputs with the following example. In the layouts, an insertion operation is indicated by the \(--\) symbol.

**Example 2** Given the string \( x = \text{TGCAGTCCC} \) and the set \( U = \{\text{CCA, TCC, CTC}\} \), the minimum number \( t \) such that \( U \) is a set of approximate 3-covers for \( x \) with distance \( t \) will be computed as:

1. **Using Hamming distance**, \( t = 1 \) and a possible layout (with cover set \( V = \{\text{TGC, GCA, GTC, CCC}\} \)) is as follows:

   \[
   \begin{array}{cccccccc}
   T & G & C & A & G & T & C & C & C \\
   T & C & C &  & C & T & C & & \\
   C & C & A &  & C &  & C & & \\
   
   \end{array}
   \]

2. **Using Levenshtein distance**, \( t = 1 \) and a possible layout (with cover set \( V = \{\text{TGC, GCA, GTC, TCCC}\} \)) is as follows:

   \[
   \begin{array}{cccccccc}
   T & G & C & A & G & T & C & C & C \\
   T & C & C &  & C & T & C & & \\
   C & C & A &  & C &  & C & & \\
   C & & C & & - & & & & \\
   \end{array}
   \]

3. **Using edit distance**, \( t = 2 \) and a possible layout (with cover set \( V = \{\text{TGC, GCA, GTC, TCCC}\} \)) is as follows:

   \[
   \begin{array}{cccccccc}
   T & G & C & A & G & T & C & C & C \\
   T & - & C & C &  & C & T & C & \\
   - & C & C &  & C &  & C & & \\
   - & - & C & T & C & & & & \\
   \end{array}
   \]

4. **Algorithm under Hamming Distance**

   In this section, we define distance as Hamming distance, which counts the number of mismatches between two strings of same length. We present an \( O(km(n - k)) \) time algorithm for solving Problem \( t_h \). As the definition of distance is specified, we can make Definition 2 more appropriate. Indeed, \( V \) is a (multi)set of \( k \)-covers for the string \( x \).

   Given a string \( x \) of length \( n \) and a set \( U = \{u_1, \ldots, u_m\} \) of strings of length \( k \), the following are some basic facts about \( U \) being a set of approximate \( k \)-covers for \( x \) with distance \( t \) generating a (multi)set \( V = \{v_i, \ldots, v_{m'}\} \) covering \( x \):

   **Fact 4** A substring of \( x \) may have a multiplicity bigger than 1 in \( V \). Moreover, \( v_1 \) is a prefix of \( x \), \( v_{m'} \) is a suffix of \( x \), and \( v_i \) concatenates or overlaps with \( v_{i+1} \) for \( 1 \leq i < m' \).

   **Fact 5** There may exist \( 1 < i < i' < m \) and \( 1 < j < j' < m' \) such that \( u_i \) generates \( v_j \) and \( u_{i'} \) generates \( v_{j'} \). (Example 2(1) shows this fact.)
Fact 6 Every element in $U$ must be used to generate at least one element in $V$, and every element in $V$ is generated by at least one element in $U$. (In Example 2(1), CCA is used to generate both GCA and CCC.)

Fact 7 A (multi)set $V$ of covers for $x$ is not unique. (For example, if $x = TCATCATCT$ and $U = \{TCGT, ATCT\}$, then $U$ is a set of approximate 4-covers for $x$ with distance 1. One of the cover sets is $V_1 = \{TCAT, ATCA, ATCT\}$ while the other is $V_2 = \{TCAT, TCAT, ATCT\}$. In general, there may be an exponential number of (multi)sets of covers for $x$.)

Fact 8 The strings $x[1..k]$ and $x[n - k + 1..n]$ are both elements of $V$.

Based on Fact 8 and Definition 2, we get Fact 9:

Fact 9 If $u_i$ is a generator for $x[1..k]$ and $u_j$ is a generator for $x[n - k + 1..n]$ for some $1 \leq i, j \leq m$, then $t \geq \max(\delta(u_i, x[1..k]), \delta(u_j, x[n - k + 1..n]))$.

The main ideas for the algorithm are clear: Fact 5 shows that it is not easy to figure out which element of $U$ generates which element of $V$; Fact 8 states that the strings $x[1..k]$ and $x[n - k + 1..n]$ are always in $V$; Further, Fact 9 implies that

$$t \geq \max(\min_{1 \leq i \leq m} \delta(u_i, x[1..k]), \min_{1 \leq i \leq m} \delta(u_i, x[n - k + 1..n]))$$

Therefore, the algorithm uses

$$d = \max(\min_{1 \leq i \leq m} \delta(u_i, x[1..k]), \min_{1 \leq i \leq m} \delta(u_i, x[n - k + 1..n]))$$

(2)

as a yardstick to find the minimum number $t$ and a (multi)set $V$ satisfying Definition 2. Initially, the algorithm initializes $d$ as in Eq.(2) and sets $d$ as the comparing criterion to obtain a (multi)set $V$ of pseudo-covers such that $\delta(u, v) \leq d$ for $u \in U$, $v \in V$. Then the algorithm tests whether this (multi)set of pseudo-covers $V$ generated by $U$ satisfies Definition 2. In order to do this, using the idea from Fact 4, the algorithm tests whether $V$ covers $x$ or not (this is done using Algorithm CoverTest), and also using the idea from Fact 6, the algorithm tests whether every element in $U$ is used as a generator or not (this is done by using a Boolean array to mark every element in $U$ that has been used). If the (multi)set of pseudo-covers $V$ satisfies Definition 2, then the algorithm returns $d$ as the minimum number $t$. Otherwise, the algorithm increases $d$ by 1, and repeats the previous tests until $V$ is found.

To illustrate the ideas, let $x = CTTATTTAA$ and $U = \{CTTA, TTAA\}$. After covering the prefix and the suffix of length 4 of $x$, we get

```
C T T A T T T A A  
C T T A       T T A A
```

and CoverTest returns FALSE since $x[5]$ is not covered. In this situation, $d$ is increased by 1 and we obtain the following layout

```
C T T A T T T A A  
C T T A       T T A A
```

with CoverTest returning TRUE.
To achieve efficiency, the following variables and data structures are used:

- An integer $n$
  $n$ is the length of $x$.

- An integer $k$
  $k \leq n$ is the length of the elements in $U$.

- An integer $m$
  $m$ is the cardinality of $U$.

- A 2-dimensional integer array $D$
  $D[i, j]$, where $1 \leq i \leq m$ and $1 \leq j \leq n - k + 1$, records the Hamming distance $\delta(u_i, x[j..j + k - 1])$. The array $D$ is called the distance table.

- A 2-dimensional Boolean array $G$
  $G[i, j]$, where $1 \leq i \leq m$ and $1 \leq j \leq n - k + 1$, records TRUE if $D[i, j] = \delta(u_i, x[j..j + k - 1]) \leq d$ where $d$ is the comparing criterion initialized as in Eq.(2); $G[i, j]$ records FALSE otherwise. The array $G$ is called the generator table.

- A global Boolean array $V$
  $V[j]$, where $1 \leq j \leq n - k + 1$, records TRUE if there exists $i$ such that $1 \leq i \leq m$ and $G[i, j] = TRUE$; $V[j]$ records FALSE otherwise. The array $V$ is used for cover testing. It records the beginning of all the pseudo-covers produced by elements in $U$.

- A Boolean array MARK
  $MARK[i]$, where $1 \leq i \leq m$, records TRUE if $u_i$ is used as a generator to construct $x$; $MARK[i]$ records FALSE otherwise.

Algorithm $th$

The algorithm consists of three steps.

**Step 1:** For $1 \leq i \leq m$ and $1 \leq j \leq n - k + 1$, use Algorithm $h$-Distance to compute $D[i, j]$ which is the Hamming distance between 1.4 and $x[j..j + k - 1]$.

**Step 2:** Initialize $d$ as in Eq.(2). For $1 \leq j \leq n - k + 1$, initialize $V[j]$ with FALSE. And for $1 \leq i \leq m$ and $1 \leq j \leq n - k + 1$, initialize $G[i, j]$ with FALSE and $MARK[i]$ with FALSE.

**Step 3:** For $1 \leq i \leq m$ and $1 \leq j \leq n - k + 1$, update $G[i, j]$, $V[j]$ and $MARK[i]$ with TRUE’s if $D[i, j] \leq d$. If there exists $1 \leq i \leq m$ such that $MARK[i] = FALSE$ or if there exist at least $k$ consecutive entries in $V$ recorded as FALSE (use Algorithm CoverTest to find out if the latter condition holds), then increase $d$ by 1 and repeat to modify table $G$, array $V$, and array $MARK$; otherwise, Algorithm $th$ returns $d$ as the minimum $t$ such that $U$ is a set of approximate $k$-covers for $x$ with distance $t$.

Note: In order to compute a layout for $x$ with minimum distance, pick up entries in $G$ that are TRUE: say, $G[i_1, j_1], \ldots, G[i_r, j_r]$ where $\{i_1, \ldots, i_r\} = \{1, \ldots, m\}$ and $1 \leq j_1 < \cdots < j_r \leq n - k + 1$. If the (multi)set $V = \{x[j_1..j_1 + k - 1], \ldots, x[j_r..j_r + k - 1]\}$ covers $x$, then $V$ is as desired. In this case, $u_{i_a}$ is a generator for $x[j_s..j_s + k - 1]$ for all $1 \leq s \leq r$. 

We now express Algorithm $t_h$ in pseudo programming language code.

**Algorithm $h$-Distance**

**input:** strings $u$ and $v$ of length $k$

**output:** Hamming distance between $u$ and $v$

```plaintext
dist ← 0
for $i ← 1$ to $k$ do
  if $u[i] = v[i]$ then $h ← 0$
  else $h ← 1$
  dist ← dist + $h$
return dist
```

**Algorithm CoverTest**

**input:** Boolean array $V$ of size $n - k + 1$

**output:** TRUE (if $V$ covers $x$) or FALSE (otherwise)

```plaintext
flag ← TRUE
i ← 1
while $i < n - k + 1$ and flag = TRUE do $j$
  $j ← i + 1$
  while $V[j] = \text{FALSE}$ and $j < n - k + 1$ do $j$
    $j ← j + 1$
  if $V[j] = \text{TRUE}$ and $j - i < k$ then
    $i ← j$
  else flag ← FALSE
return flag
```

**Algorithm $t_h$**

**input:** string $x$ and set $U = \{u_1, \ldots, u_m\}$ of strings where $0 < |u_1| = \cdots = |u_m| \leq |x|$

**output:** the minimum number $t$ such that $U$ is a set of approximate $|u_1|$-covers for $x$ with Hamming distance $t$

```plaintext
n ← |$x$|
k ← |$u_1$|

// Step 1: Compute $D$
for $i ← 1$ to $m$ do
  for $j ← 1$ to $n - k + 1$ do
    $D[i,j] ← h$-Distance($u_i, x[j..j+k-1]$)

// Step 2:
// Initialize $d$
$fmin ← \min_{1≤i≤m} D[i,1]$
$lmin ← \min_{1≤i≤m} D[i,n - k + 1]$
$d ← \max(fmin, lmin)$
// Initialize $G$, $V$ and $MARK$
for $j ← 1$ to $n - k + 1$ do
  $V[j] ← \text{FALSE}$
```

for $i \leftarrow 1$ to $m$ do
    \[ G[i, j] \leftarrow \text{FALSE} \]
    \[ \text{MARK}[i] \leftarrow \text{FALSE} \]

// Step 3: Process

find $\leftarrow$ \text{FALSE}

while find $= \text{FALSE}$ do
    for $j \leftarrow 1$ to $n - k + 1$ do
        for $i \leftarrow 1$ to $m$ do
            if $D[i, j] \leq d$ then
                \[ G[i, j] \leftarrow \text{TRUE} \]
                \[ V[j] \leftarrow \text{TRUE} \]
                \[ \text{MARK}[i] \leftarrow \text{TRUE} \]
            end if
        end for
    end for
    if \text{MARK}[i] = \text{TRUE} for all $1 \leq i \leq m$ and \text{CoverTest}(V) = \text{TRUE} then
        find $\leftarrow \text{TRUE}$
    end if
    $d \leftarrow d + 1$
end while

$t \leftarrow d$

return $t$

Let us now determine the complexity of Algorithm $t_h$.

**Theorem 2** On input string $x$ of length $n$ and set $U$ of $m$ strings of length $k$, Algorithm $t_h$ terminates with the minimum $t$ such that $U$ is a set of approximate $k$-covers for $x$ with distance $t$. Moreover, Algorithm $t_h$ solves Problem $t_h$ in $O(km(n - k))$ time.

**Proof.** Step 1 of Algorithm $t_h$ has two nested loops. They do the computation of the distance table $D$ by using Algorithm $h$-Distance that requires $O(k)$ time for each entry. Thus, the total complexity of Step 1 is $O(km(n - k))$ time. The initialization in Step 2 requires $O(m(n - k))$ time. The dominant term in the time complexity of Step 3 is the while loop which is executed at most $k + 1$ times since $t$ should be less than or equal to $k$. This loop has two nested for loops: the first is executed $n - k + 1$ times, and the second $m$ times. Also, the while loop calls Algorithm CoverTest which requires $O(n - k)$ time. Thus, the total complexity of Step 3 is $O(km(n - k))$. Hence, the overall complexity of Algorithm $t_h$ is $O(km(n - k))$ time.

We now illustrate Algorithm $t_h$ with the following example.

**Example 3** Given the string $x = \text{GCATCATGTCTT}$ of length 12 and the set $U = \{\text{ACAT, ATCA, TCGT}\}$, Algorithm $t_h$ computes the minimum number $t$ such that $U$ is a set of approximate 4-covers for $x$ with distance $t$ as $t = 2$. A possible layout is

```
  G C A T
  A C A T
  A T C A
  A T C A
```

5. Algorithm under Levenshtein Distance

In this section, we define distance as Levenshtein distance. We give an $O(mn^2)$ time algorithm to solve Problem $t_l$. The difference between Levenshtein distance and Hamming distance is that the tranformation restrictions are relaxed allowing substitutions, insertions and deletions.

Given a string $x$ and a set $U = \{u_1, \ldots, u_m\}$ of $k$-strings, in addition to Facts 4-7 of Section 4, the following are some basic facts about $U$ being a set of approximate $k$-covers for $x$ with distance $t$ generating a (multi)set $V = \{v_1, \ldots, v_m\}$ covering $x$:

**Fact 10** The lengths of elements in $V$ are not necessarily equal. (Example 2(2) shows this fact.)
Based on Fact 6, we get Fact 11:

**Fact 11** The relation

\[ t \geq \max_{1 \leq i \leq m} \left( \min_{v \in V} \delta(u_i, v) \right) \]

holds.

The main ideas for the algorithm are as follows: Fact 10 implies that Facts 8-9 do not hold for Levenshtein distance since the lengths of \( v_1 \) and \( v_m' \) are not known. However, Fact 11 gives a relation between \( t \) and the elements in \( U \) and \( V \). Thus, instead of using Eq.(2) as the comparing criterion, the algorithm uses the following equation to initialize \( d \):

\[ d = \max_{1 \leq i \leq m} \left( \min_{v \in V} \delta(u_i, v) \right) \]  

(3)

Distance computing is more complicated in the Levenshtein version than in the Hamming distance version since deletions and insertions are also allowed. Here we use Algorithm *l-Distance* explained in more details below.

Cover length computing is also more complicated in the Levenshtein version than in the Hamming distance version since the lengths of elements in \( V \) may be different as stated in Fact 10. The algorithm computes in two steps all cover lengths \( |v| \) for \( v \in V \). First, the algorithm uses Algorithm *CoverLength* to compute \( |v| \) without considering insertions at the beginning of \( u \) when transforming \( u \) into \( v \). For example,

\[
\begin{array}{cccccccccc}
A & C & G & C & & & & & & & & & & & \\
C & G & & & G & C & A & C & T & & & & & & \\
\end{array}
\]

ACGC through the deletion of a C generates the cover AGC of length 3; CGGC generates the cover CGAGC of length 5 through the insertion of an A; and AACT generates the cover AACT of length 4. However, \( x[9] \) is not covered. Second, the algorithm takes care of the insertions at the beginning of \( u \). If positions \( x \) exist separating two consecutive pseudo-covers \( v_i \) and \( v_{i+1} \) generated by \( u \) and \( u' \) respectively, then a gap exists between \( v_i \) and \( v_{i+1} \). In such situations where \( \delta(u', v_{i+1}) < \delta(u, u_i) \), the algorithm uses insertion operations to minimize the gap. Every insertion makes the distance \( \delta(u', v_{i+1}) \) (or \( d' \)) increase by 1. The algorithm repeats this operation until \( d' \) equals \( d \). While cover testing, if a gap still exist then the algorithm increases \( d \) by 1 and repeats to get rid of the gap. Referring the above example, we get

\[
\begin{array}{cccccccccc}
A & C & G & C & & & & & & & & & & & \\
C & G & & & & G & C & & & & A & A & C & T \\
\end{array}
\]

The following variables and data structures are used:

- An integer \( n \)
  
  \( n \) is the length of \( x \).

- An integer \( k \)
  
  \( k < n \) is the length of the elements in \( U \).
• An integer \( m \)
  \( m \) is the cardinality of \( U \).

• 2-Dimensional global integer arrays \( D_1, \ldots, D_m \)
  For \( 1 \leq h \leq m \), array \( D_h \) corresponds to the dynamic programming array of size \((n+1) \times (k+1)\) for computing the distance between \( x \) and \( u_h \) according to Algorithm \( l\)-Distance. In particular, \( D_h[i,k] \) is the distance between a suffix of \( x[1..i] \) and \( u_h \). The arrays \( D_1, \ldots, D_m \) are called the distance tables.

• 2-Dimensional global integer arrays \( L_1, \ldots, L_m \)
  For \( 1 \leq h \leq m \), array \( L_h \) is of size \((n+1) \times (k+3)\). The first \( k+1 \) columns of \( L_h \) correspond to the \( k+1 \) columns of the distance table \( D_h \). The \((k+2)\)th column of \( L_h \) is computed with Algorithm \( \text{CoverLength} \). The last column of \( L_h \) records the number of insertions at the beginning of generator \( u_h \). The arrays \( L_1, \ldots, L_m \) are called the length tables.

• A 2-dimensional integer array \( G \)
  \( G[i,j] \), where \( 1 \leq i \leq m \) and \( 1 \leq j \leq n \), records the cost for transforming \( u_i \) into the suffix of \( x[1..j] \) generated by \( u_i \) if that cost is smaller than or equal to \( d \) where \( d \) is the comparing criterion initialized as in Eq.(3); \( G[i,j] \) records -1 otherwise. The array \( G \) is called the generator table.

• A global Boolean array \( M \)
  \( M[i] \), where \( 1 \leq i \leq n \), records TRUE if \( x[i] \) has been covered by a pseudo-cover; \( M[i] \) records FALSE otherwise.

Algorithm \( t_i \)

The algorithm consists of four steps.

Step 1: For \( 1 \leq h \leq m \), use Algorithm \( l\)-Distance to compute table \( D_h \) for the Levenshtein distance between \( x \) and \( u_h \) when spaces are not charged for at the beginning and end of \( u_h \). More precisely, for \( 0 \leq i \leq n \) and \( 0 \leq j \leq k \), use Eq. (4) to compute \( D_h[i,j] \).

Step 2: For \( 1 \leq h \leq m \), copy the columns of table \( D_h \) into the corresponding columns of table \( L_h \) and initialize the last two columns of table \( L_h \) with zeros. Next, for \( 1 \leq i \leq n \), use Algorithm \( \text{CoverLength} \) to compute \( L_h[i,k+1] \) which is the length of the suffix of \( x[1..i] \) generated by \( u_h \) (call \( \text{CoverLength}(i,k,D_h) \)). To do this, the call \( \text{CoverLength}(i,k,D_h) \) starts at \( D_h[i,k] \) counting the number of arrows (\& highest priority) and (↑ next priority) until Column 0 of \( D_h \) is hit.

Step 3: First, initialize table \( G \) with −1’s and array \( M \) with FALSE’s. Second, initialize the comparing criterion \( d \) with \( d = \max_{1 \leq h \leq m} (\min_{1 \leq i \leq k} D_h[i,k]) \).

Step 4: For \( 1 \leq h \leq m \) and \( 1 \leq i \leq n \), compare \( D_h[i,k] \) with \( d \). If \( D_h[i,k] \leq d \), then save the value \( D_h[i,k] \) in table \( G \) as \( G[h,i] \). Then, compute the length \( l \) of the suffix of \( x[1..i] \) whose distance with \( u_h \) is bounded by \( d \), and update \( L_h[i,k+2] \). Next, update \( M[j] \) with TRUE for \( i - l < j \leq i \). If there exists \( 1 \leq i \leq n \) such that \( M[i] = \text{FALSE} \), then \( x[i] \) is not covered and increase \( d \) by 1 repeating Step 4 to modify table \( G \) and array \( M \). Otherwise, return \( d \) as the minimum number \( t \) such that \( U \) is a set of approximate \( k \)-covers for \( x \) with distance \( t \).

Note: In order to compute a layout for \( x \) with minimum distance, pick up entries in \( G \) that are not −1: say, \( G[i_1,\ldots,i_l], \ldots, G[i_r,i_r] \) where \( \{i_1,\ldots,i_r\} = \{1,\ldots,m\} \) and \( 1 \leq j_1 < \cdots < j_r \leq n \). Put \( l_s = L_{i_s}[j_s,k+1] + L_{i_s}[j_s,k+2] \) for all \( 1 \leq s \leq r \)

\( (L_{i_s}[j_s,k+2] \) is the number of insertions that can be added if needed at the beginning of \( u_{i_s} \) in the layout). If the (multi)set
\[ V = \{x[j_1 - l_1 + 1..j_1],...,x[j_r - l_r + 1..j_r]\} \]

covers \( x \), then \( V \) is as desired. In this case, \( u_{is} \) is a generator for \( x[j_s - l_s + 1..j_s]\) for all \( 1 \leq s \leq r \).

The well-known paper by Needleman and Wunsch [12] is an important contribution for computing the distance between two strings \( x \) and \( u \) relative to a measure \( \delta \). Finding the best alignment between these two strings can be solved efficiently by dynamic programming. Let us now describe a variation of this basic algorithm that will ignore end spaces in \( u \). In order to do so, a \( D \) table of size \( (|x| + 1) \times (|u| + 1) \) is used. We can initialize the first column with zeros, and by doing this we will be forgiving spaces before the beginning of \( u \). Initially, \( D[i,0] = 0 \) for all \( 0 \leq i \leq |x| \), and \( D[0,j] = D[0,j - 1] + 1 \) for all \( 1 \leq j \leq |u| \). We can compute all the entries of the \( D \) table in \( O(|x| |u|) \) time by the following recurrence:

\[
D[i, j] = \min \begin{cases} 
D[i, j - 1] + 1 \\
D[i - 1, j - 1] + p[i, j] \\
D[i - 1, j] + 1 
\end{cases} 
\tag{4}
\]

where scoring function \( p[i, j] = 0 \) if \( x[i] = u[j] \), and \( p[i, j] = 1 \) if \( x[i] \neq u[j] \). We can look for the minimum in the last column, and by doing this we will be forgiving spaces after the end of \( u \). Algorithm \( l \)-Distance fills \( D \) as explained where for \( 0 \leq i \leq |x| \) and \( 0 \leq j \leq |u| \), entry \( D[i, j] \) records the minimum cost of transforming a suffix of \( x[1..i] \) into \( u[1..j] \).

**Algorithm \( l \)-Distance**

**input:** strings \( x \) and \( u \)

**output:** Levenshtein distance between \( x \) and \( u \) when spaces are not charged for at the beginning of \( u \) and end of \( u \)

\[
n \leftarrow |x| \\
k \leftarrow |u|
\]

for \( i \leftarrow 0 \) to \( n \) do

\[
D[i, 0] \leftarrow 0
\]

for \( j \leftarrow 0 \) to \( k \) do

\[
D[0, j] \leftarrow j
\]

for \( i \leftarrow 1 \) to \( n \) do

\[
D[i, j] \leftarrow \min(D[i, j - 1] + 1, D[i - 1, j - 1] + p[i,j],D[i - 1, j] + 1)
\]

return \( \min_{1 \leq i < n} D[i, k] \)

We described Algorithm \( l \)-Distance which computes the distance table \( D \) for the Levenshtein distance between two strings \( x \) and \( u \) when spaces are ignored at either end of \( u \). Here we describe Algorithm \( CoverLength \) which is recursive. Among other things, the call \( CoverLength \ [|x|, |u|, D] \) constructs an optimal alignment between \( x \) and \( u \) which is given in a pair of vectors \( \text{align}_x \) and \( \text{align}_u \) that hold in the positions \( 1..\text{len} \) the aligned characters, which can be either spaces or symbols from the strings. The variables \( \text{len}, \text{clen}, \text{align}_x \) and \( \text{align}_u \) are treated as globals in the code.

**Algorithm \( CoverLength \)**

**input:** indices \( i, j \), and table \( D \) given by Algorithm \( l \)-Distance

**output:** alignment in \( \text{align}_x, \text{align}_u \), length of the alignment in \( \text{len} \), and length of the suffix of \( x[1..i] \) generated by \( u \) in \( \text{clen} \)
if \( i = 0 \) or \( j = 0 \) then
\[
\text{clen} \gets 0
\]
\[
\text{len} \gets 0
\]

// Substitution from \( u \) to \( x \)
else if \( i > 0 \) and \( j > 0 \) and \( D[i, j] = D[i - 1, j - 1] + p(i, j) \) then

CoverLength\((i - 1, j - 1, D)\)
\[
\text{len} \gets \text{len} + 1
\]
\[
\text{align}_x[\text{len}] \gets x[i]
\]
\[
\text{align}_u[\text{len}] \gets u[j]
\]
\[
\text{den} \gets \text{den} + 1
\]

// Insertion from \( u \) to \( x \)
else if \( i > 0 \) and \( j > 0 \) and \( D[i, j] = D[i - 1, j] + 1 \) then

CoverLength\((i - 1, j, D)\)
\[
\text{len} \gets \text{len} + 1
\]
\[
\text{align}_x[\text{len}] \gets x[i]
\]
\[
\text{align}_u[\text{len}] \gets --
\]
\[
\text{den} \gets \text{den} + 1
\]

// Deletion from \( u \) to \( x \)
else

CoverLength\((i, j - 1, D)\)
\[
\text{len} \gets \text{len} + 1
\]
\[
\text{align}_x[\text{len}] \gets --
\]
\[
\text{align}_u[\text{len}] \gets u[j]
\]

We now describe Algorithm \( t_f \) in pseudo programming language code.

Algorithm \( t_f \)

input: string \( x \) and set \( U = \{u_1, \ldots, u_m\} \) of strings where \( 0 < |u_1| = \cdots \leq |x| \)

output: the minimum number \( t \) such that \( U \) is a set of approximate \( |u_1| \)-covers for \( x \) with Levenshtein distance \( t \)

\[
n \gets |x|
\]
\[
k \gets |u_1|
\]

// Step 1: Compute \( D_1, \ldots, D_m \)
for \( h \gets 1 \) to \( m \) do

\( l\)-Distance\((x, u_h)\)

for \( i \gets 0 \) to \( n \) do

for \( j \gets 0 \) to \( k \) do

// Copy \( D \) computed by the call \( l\)-Distance\((x, u_h)\) to \( D_h \)

\( D_h[i, j] \gets D[i, j] \)

// Step 2: Compute \( L_1, \ldots, L_m \)
for \( h \gets 1 \) to \( m \) do

for \( i \gets 0 \) to \( n \) do

\( L_h[i, k + 1] \gets 0 \)

\( L_h[i, k + 2] \gets 0 \)

for \( j \gets 0 \) to \( k \) do
\[ Lh[i,j] \leftarrow Dh[i,i] \]

for \( h \leftarrow 1 \) to \( m \) do
  for \( i \leftarrow 1 \) to \( n \) do
    CoverLength(\( i, k, D_h \))
    // The length of the cover generated by \( u_h \) and ending at position \( i \) is
    // computed in \( clen \)
    \( Lh[i, k + 1] \leftarrow clen \)

// Step 3:
// Initialize \( G \) and \( M \)
for \( j \leftarrow 1 \) to \( n \) do
  \( M[j] \leftarrow \text{FALSE} \)
  for \( I \leftarrow 1 \) to \( m \) do
    \( G[i,j] \leftarrow -1 \)

// Initialize \( d \)
\( d \leftarrow \max_{1 \leq h \leq m} (\min_{1 \leq i \leq n} D_h[i, k]) \)

Step 4: Process
\( \text{find} \leftarrow \text{FALSE} \)
while \( \text{find} = \text{FALSE} \) do
  // Compute \( G \) and \( M \)
  for \( h \leftarrow 1 \) to \( m \) do
    for \( i \leftarrow 1 \) to \( n \) do
      temp \( \leftarrow D_h[i, k] \)
      if \( temp \leq d \) and \( G[h, i] = -1 \) then
        \( G[h, i] \leftarrow temp \)
      // Compute the length \( l \) of the longest cover ending at position \( i \) and generated by \( u_h \)
      \( l \leftarrow Lh[i, k + 1] + (d - temp) \)
      // Update \( Lh \)
      if \( Lh[i, k + 1] \neq l \) then \( Lh[i, k + 2] d-temp \)
      // Update \( M \)
      for \( j \leftarrow i - l + 1 \) to \( i \) do
        \( M[j] \leftarrow \text{TRUE} \)

  // Cover test
  \( i \leftarrow 1 \)
  cover \( \leftarrow \text{TRUE} \)
  while \( i \leq n \) and cover = \( \text{TRUE} \) do
    if \( M[i] = \text{FALSE} \) then cover \( \leftarrow \text{FALSE} \)
    else \( i \leftarrow i + 1 \)
    if cover = \( \text{FALSE} \) then \( d \leftarrow d + 1 \)
    else find \( \leftarrow \text{TRUE} \)

\( t \leftarrow d \)
return \( t \)

We now analyze the complexity of Algorithm \( t_l \).

**Theorem 3** On input string \( x \) of length \( n \) and set \( U \) of \( m \) strings of length \( k \), Algorithm \( t_l \) terminates with the minimum \( t \) such that \( U \) is a set of approximate \( k \)-covers for \( x \) with distance \( t \). Moreover, Algorithm \( t_l \) solves Problem \( t_l \) in \( O(mn^2) \) time.
Proof. For $1 \leq h \leq m$, Step 1 does the computation of the distance table $D_h$ using Algorithm $l$-Distance. The call $l$-Distance$(x, u_h)$ requires $O(kn)$ time and thus, the complexity of Step 1 is $O(kmn)$ time.

For $1 \leq h \leq m$, Step 2 does the computation of the first $k + 2$ columns of the length table $L_h$ along with the initialization of its last column. Among other things, for $1 \leq i \leq n$, the call CoverLength$(i, k, D_h)$ does the construction of the alignment between $x[1..i]$ and $u_h$ (given the already filled array $D_h$) in time $O(len)$, where $len$ is the size of the alignment, which is $O(i + k)$. The call CoverLength$(i, k, D_h)$ also computes in $clen$ the length of the cover generated by $u_h$ and ending at position $i$ of $x$. This computation also requires $O(i + k)$ time. Thus, the total complexity of Step 2 is $O(mnk^2)$ time.

The initializations of $G$, $M$ and $d$ in Step 3 take $O(mn)$ time. The while loop in Step 4 is executed at most $k + 1$ times. Each pass through the loop updates $G$ and $M$ in $O(mn)$ time, and also tests for the covering of $x$ in $O(n)$ time. Thus, the total complexity of Step 4 is $O(kmn)$. Therefore, the total complexity of Algorithm $t_1$ is $O(mnk^2)$ time.

We end this section with the following example.

**Example 4** Given the string $x = CTGTCAACT$ of length 9 and the set $U = \{ACT, CTT, AAC\}$, Algorithm $t_1$ computes the minimum number $t$ such that $U$ is a set of approximate 3-covers for $x$ with distance $t$ as $t = 1$. A possible layout is as follows:

```
C T G T C A A C T
C T - T - A A C
-         A C T
```

6. Algorithm under Edit Distance

In edit distance, the operations allowed are insertions and deletions; substitutions are not allowed. Algorithm $t_1$ can be used to solve Problem $t_e$ by disabling substitution operations. Indeed, we modify the scoring function in Algorithm $l$-Distance as follows: if $x[i] = u[j]$, let $p[i, j] = 0$; and if $x[i] \neq u[j]$, let $p[i, j] = +\infty$.

The complexity of Algorithm $t_e$ is stated in the next theorem.

**Theorem 4** On input string $x$ of length $n$ and set $U$ of $m$ strings of length $k$, Algorithm $t_e$ terminates with the minimum $t$ such that $U$ is a set of approximate $k$-covers for $x$ with distance $t$. Moreover, Algorithm $t_e$ solves Problem $t_e$ in $O(mnk^2)$ time.

We illustrate Algorithm $t_e$ with the following example.

**Example 5** Given the string $x = GCATCATGTCTT$ of length 12 and the set $U = \{ACAT, ATCA, TCGT\}$, Algorithm $t_e$ computes the minimum number $t$ such that $U$ is a set of approximate 4-covers for $x$ with distance $t$ as $t = 2$. A possible layout is as follows:

```
G C A T C A T G T C T T
- A C A T
A T C A T C G T T C G - T
```

The Hamming, Levenshtein and edit distances can be generalized by using a penalty matrix. Such a matrix specifies the substitution cost for each pair of characters and the insertion/deletion cost for each character. The simplest matrix assumes costs of $g_1$ for the substitutions and costs of $g_2$ for the insertions/deletions. Algorithm $t_1$ can easily be generalized by using for instance Eq.(5) described as follows:
$D[i,j] = \min \begin{cases} 
D[i,j-1] + g_2 \\
D[i-1,j-1] + p[i,j] \\
D[i-1,j] + g_2.
\end{cases}$

where scoring function $p[i,j] = 0$ if $x[i] = u[j]$, and $p[i,j] = g_1$ if $x[i] \neq u[j]$.

Acknowledgements
This material is based upon work supported by the National Science Foundation under Grant No. CCF-0207673. We thank the referee of a preliminary version of this paper for his/her very valuable comments and suggestions.

References:

Notes:
a (Multi)set of pseudo-covers: A (multi)set $V$ that is generated by $U$, but unproved to cover $x$ is called a (multi)set of pseudo-covers for $x$. 