Note

$k$-difference matching in amortized linear time for all the words in a text

Cinzia Pizzi

Department of Information Engineering, University of Padova, Italy

**A R T I C L E I N F O**

Article history:
Received 28 March 2008
Received in revised form 22 November 2008
Accepted 8 December 2008
Communicated by M. Crochemore

Keywords:
String-matching problem
Words with mismatches

**A B S T R A C T**

Given a text $x$ of length $n$, we study the problem of solving the $k$-difference problem for all the words, either with fixed or variable length, taken from the text itself. The result finds its application in pattern discovery in biosequences where over- or under-represented words are extracted from the input sequences. The proposed algorithm runs in amortized linear time per word. This improves the complexity obtained by applying well-known algorithms to each of the $O(n)$ fixed length words or $O(n^2)$ variable length words in $x$ by factor of $k$, $\sqrt{k \log k}$, or $\sqrt{m \log m}$, depending on the chosen algorithm. The space required is $O(n)$ if we just count the occurrences, or $O(n^2)$ if we also store the positions. This second scenario can be used as the basis for other applications, such as searching gapped factors with mismatches or approximate pattern matching extended to any word.

© 2008 Elsevier B.V. All rights reserved.

1. Introduction

The problem of approximate string matching is widely studied in Computer Science and it finds application in several domains, including Computational Biology, Signal Processing, Text Retrieval, Data Mining, and Pattern Recognition.

Depending on the distance used to measure the “approximation” several solutions have been proposed. Two of the most popular distances are the edit distance, where insertion, deletion and mismatches are allowed (see [10] for a detailed overview of algorithms to solve this problem), and Hamming distance, where only mismatches, or differences, are allowed. For $k$-difference problem the best known algorithm finds all occurrences of a word $w$ in a text $x$ of length $n$ in $O(n \sqrt{k \log k})$ time [4]. Other well-known fast algorithms achieve $O(nk)$ [9], and $O(n \sqrt{m \log m})$ [1], where $m$ is the length of $w$.

Our problem is a specific case of $k$-difference problem, where we want to find all the occurrence with mismatches, or just their number, for all the words that occur in the text $x$ at least once exactly. This context emerges for example in pattern discovery in biological applications [6], when the signals occur at least once exactly in the input sequence. Pattern discovery in bio-sequences is often based on measures of over-representation such as $z$-scores, and $p$-values [2,11,12]. These require the computation of the actual number of occurrences of the strings in the text to compare it with the expected number.

One way to compile such score tables is to apply to every word one of the algorithms known from the literature [1,4,9]. For example, applying the algorithm in [4] the time complexity will be $(n^2 \sqrt{k \log k})$, for fixed length words, and $(n^3 \sqrt{k \log k})$, for variable length words.

In Section 2 we present an algorithm that performs the same task in $O(n^2)$ and $O(n^3)$ respectively, hence amortized linear time per word. Section 3 describes a basic implementation for fixed length words that uses $O(n^2)$ space, and a variant that computes the required output only once for each different word. Although the asymptotic complexity remains the same, in practice the latter allows for space savings. Finally, in Section 4 we describe three applications of our algorithm: (i) the computation of $z$-scores through a variant of the basic algorithm that uses only $O(n)$ space; (ii) searching gapped factors with mismatches using as a basis the indexes described in Section 2; (iii) classical $k$-difference problem, where we provide an input query that can be any word, not necessarily occurring in the text.
The proof descends from the preceding analysis.

2. Algorithm description

Given a text \( x = x_1 \ldots x_n \) of length \( n \), we denote \( x(s, e) \), with \( s \leq e \), the segment of the text \( x_s x_{s+1} \ldots x_e \). The length of \( x(s, e) \) is \( m = e - s + 1 \). For a fixed length \( m \), \( \mathcal{L}_k(s, m) \) is the set of the starting positions of occurrences of the string \( w = x(s, s + m - 1) \) in \( x \) with exactly \( k \) mismatches:

\[
\mathcal{L}_k(s, m) = \{ p : d(x(s, s + m - 1), x(p, p + m - 1)) = k \}
\]

In the following section we introduce a simple but effective algorithm that given \( x \) and \( m \) computes in overall \( O(n^2) \) time \( \mathcal{L}_k(s, m) \) for all \( s \), \( \forall k \).

2.1. Basic algorithm for fixed length

Our approach to solve the problem for fixed length words follows the philosophy of [3] by computing the number of occurrences of \( x(s + 1, s + m) \) from those of \( x(s, s + m - 1) \) sliding a window of size \( m \) along the text. In [3] this approach allows us to compute the expected number of occurrences under the i.i.d. hypothesis in amortized constant time per word. However, for the problem of counting we need to examine the positions in which the words occur with any number of mismatches, hence the amortized time per word will be linear rather than constant.

Given the set \( \{ \mathcal{L}_k(s, m) \}, 0 \leq k \leq m \) of occurrences with mismatches of the word of length \( m \) starting at position \( s \), we compute the sets \( \mathcal{L}_k(s + 1, m) \) for the next word of length \( m \) in \( x \) in two steps:

1. from the sets \( \mathcal{L}_k(s, m) \), we compute the sets \( \mathcal{L}_k(s, m + 1) \) for the word obtained by a one symbol extension to the right.
2. from the sets \( \mathcal{L}_k(s, m + 1) \), we compute the sets \( \mathcal{L}_k(s + 1, m) \), for the word obtained by a one symbol drop to the left.

The key observation is that to compute \( \mathcal{L}_k(s, m + 1) \) for all \( k \) we consider the positions \( p \in \mathcal{L}_k(s, m) \), and compare the symbol \( x_{p+m} \) with \( x_{p+m+1} \). If they are equal, then \( p \) belongs to the set \( \mathcal{L}_k(s, m + 1) \), otherwise it belongs to the set of positions \( \mathcal{L}_{k+1}(s, m + 1) \), as shown in Fig. 1.

Similarly, for each position \( p \in \mathcal{L}_k(s, m + 1) \) we compare \( x_s \) and \( x_p \), If they are equal, then \( p + 1 \) belongs to the set \( \mathcal{L}_k(s + 1, m) \), otherwise it belongs to the set of positions \( \mathcal{L}_{k+1}(s + 1, m) \), as shown in Fig. 2.

**Theorem 2.1.** Given a word \( x(s, s + m - 1) \), and the sets \( \mathcal{L}_k(s, m) \), for \( k = 0 \) to \( m \), position \( p + 1 \in \mathcal{L}_k(s + 1, m) \), for \( k = 0 \) to \( m \), if and only if one of the following conditions holds:

(i) \( p \in \mathcal{L}_k(s, m) \) and \( x_p = x_s \) and \( x_{p+m} = x_{s+m} \)
(ii) \( p \in \mathcal{L}_k(s, m) \) and \( x_p \neq x_s \) and \( x_{p+m} \neq x_{s+m} \)
(iii) \( p \in \mathcal{L}_{k-1}(s, m) \) and \( x_p = x_s \) and \( x_{p+m} \neq x_{s+m} \)
(iv) \( p \in \mathcal{L}_{k+1}(s, m) \) and \( x_p \neq x_s \) and \( x_{p+m} = x_{s+m} \)

**Proof.** The proof descends from the preceding analysis.

When we place the sliding window at position \( s \) the cardinality of \( \mathcal{L}_k(s, m) \) gives the number of occurrence of \( x(s, s + m - 1) \) with \( k \) mismatches. The actual positions can also be retrieved from the same set.
ALGORITHM N2()
1 read input text string x, and string length m
2 compute row 1 with classic k-mismatch algorithm
3 copy row 0 in column 0 (the matrix is symmetric)
4 for i ← 1 to n − m − 1
5    do
6        for j ← i + 1 to n − m − 1
7            do
8                $M_{i,j} \leftarrow M_{i-1,j-1}$
9                if $x_{i+m-1} \neq x_{j+m-1}$
10                   then $M_{i,j} \leftarrow M_{i,j} + 1$
11                   if $x_{i-1} \neq x_{j-1}$
12                      then $M_{i,j} \leftarrow M_{i,j} - 1$
13                      $M_{j,i} \leftarrow M_{i,j}$

Fig. 3. Pseudocode for quadratic time and space algorithm.

Example. Let $x = aabbabab$, $m = 4$ and $k = 2$. For the first word $x(0, 3) = aabb$ the sets are: $\mathcal{L}_0(0, 4) = \{\}$, $\mathcal{L}_1(0, 4) = \emptyset$, $\mathcal{L}_2(0, 4) = \{1, 3, 4, 5, 6\}$, and $\mathcal{L}_3(0, 4) = \{2\}$. The number of occurrences of aabb with 2 mismatches is $|\mathcal{L}_2(0, 4)| = 5$. Moreover, $x_1 = x_0 = a$, and $x_{i+m} = x_4 = a$. To compute the sets for the shifted word $x(1, 4) = abba$, we consider each position $p$. For example, if $p = 3$ we have $x_p = b$, and $x_{p+m} = b$. Since $p \in \mathcal{L}_2(0, 4)$, $x_0 \neq x_p$ and $x_{i+m} \neq x_{p+m}$, case ii) of Theorem 1 holds, and $p + 1 = 4 \in \mathcal{L}_2(1, 4)$. It is easy to verify that the Hamming distance between abba and abab is indeed 2.

2.2. Varying the word length

The algorithm described above computes the occurrences for all the words of a given fixed length $m$ in $O(n^2)$ time. The substrings of a text of length $n$ have lengths varying from 1 to $n$. Calling the previous algorithm passing the length of the strings as a parameter gives $O(n^3)$ complexity. However, since there are $O(n^2)$ substrings, the amortized time remains linear.

3. Implementations

We first show an implementation of the algorithm that uses $O(n^2)$ space and runs in amortized linear time per word. Next we show a variant of the algorithm that reduces the actual time and space to $O(tm)$, where $t$ is the number of different words of size $m$ in $x$. The worst case remains $O(n^2)$, but in practical cases this version could allow substantial savings.

3.1. Basic implementation

To implement the algorithm we introduce a table $M$ of size $n \times n$ to store the content of the sets $\mathcal{L}_k(s, m)$ for $k = 0 \ldots m$ and $s = 1 \ldots n - m + 1$. Each row (column) $i$ is associated with the string $x(i, i + m - 1)$. The entry $M_{i,j}$ contains the number of mismatches between $x(i, i + m - 1)$ and $x(j, j + m - 1)$.

Fig. 3 shows the algorithm pseudocode. After the initial $O(kn)$ set up (lines 1–3), we proceed by filling $M$ row by row. Indeed, since $M$ is symmetric, we just need to compute $M_{i,j}$ for $j \geq i$. As seen for the computation of $\mathcal{L}$ to fill row $i$ we need to know the values of row $i - 1$. In particular, the value of $M_{i,j}$ is initially set to $M_{i-1,j-1}$ (line 8). Next, if $x_{i+m-1} \neq x_{j+m-1}$, sliding the window we acquire a mismatch, and $M_{i,j}$ is increased (lines 9–10). If $x_{i-1} \neq x_{j-1}$, sliding the window we loose a mismatch, and $M_{i,j}$ is decreased (lines 11–12). Finally, the symmetric position is filled (line 13). The time complexity is driven by the two nested For cycles, hence it is $O(n^2)$.

The space needed is $O(n^2)$ to store $M$. For any word $x(i, i + m - 1)$ we get the positions $j$ for which $M_{i,j} \leq k$ by a simple scan of row $i$. This can be done during the construction of $M$ without affecting the total complexity. Table 1 shows an example.

3.2. Optimal implementation

The preceding implementation has the drawback to recompute the set of occurrences for the same word if it occurs at several different positions in $x$. This is necessary because to compute row $i$ we need to use the values of row $i - 1$. Here we describe a simple modification of the algorithm to compute and store the data only once for each different word.

First we build an array $A$ of size $n$ such that $A[j]$ holds the index of the first occurrence of $x(j, j + m - 1)$ in the text $x$. This can be done in linear time by established indexing techniques for exact words [8]. For each string $x(i, i + m - 1)$, we look up the value of $A[i]$ before computing its occurrences. If $A[i] = i$ this is the first occurrence of $x(i, i + m - 1)$ and we run the algorithm as before. If $A[i] \neq i$ we already compute the occurrences for this word, hence we skip to the next index. Indeed we continue to skip positions until we find an index $p > i$ such that $A[p] = p$. At this step we need to retrieve the
Table 1
Example of computation of $M$ for $x = aabbababaa$ and $m = 3$. At row 3 we are computing the occurrences with mismatches for the string $x(3, 5) = \text{bab}$. Since $M_{2, 3} = 2, x_2 = x_3$ and $x_4 \neq x_5$ we have $M_{1, 4} = M_{2, 3} + 1 = 3$. Indeed $\text{bab}$ occurs with 3 mismatches at position 4 (ab).

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>a</th>
<th>b</th>
<th>b</th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>aab</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>abb</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>bba</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>bab</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Fig. 4. Pseudocode for counting the co-occurrences with mismatches of all the words of length $m$ in $x$ using linear space.

```
ONLYCOUNT_NSPACE()
1   read input text string x
2   initialize array $P_{old}$ and $P_{cur}$
3   fill array $P_{old}$ with classic k-mismatch algorithm
4   for $i \leftarrow 1$ to $n - m - 1$
5       do
6           $P_{cur}[i] \leftarrow 0$
7           for $j \leftarrow i + 1$ to $n - m - 1$
8              do
9                  $P_{cur}[j] \leftarrow P_{old}[j - 1]$
10                 if $x_{i+m-1} \neq x_{j+m-1}$
11                     then $P_{cur}[j] \leftarrow P_{cur}[j] + 1$
12                 if $x_{i-1} \neq x_{j-1}$
13                     then $P_{cur}[j] \leftarrow P_{cur}[j] - 1$
14                 if $P_{cur}[j] \leq k$
15                     then
16                         $N_{occ}[i] \leftarrow N_{occ}[i] + 1$
17                         $N_{occ}[j] \leftarrow N_{occ}[j] + 1$
```

preceding row, and this can be done by looking up row $A[p - 1]$ which has been computed before. The total space and time of this variant is $O(tm)$, where $t$ is the number of different words in the text.

Example. For $x = aabbababaa$, $m = 3$, we have $A = [0, 1, 2, 3, 4, 3, 4, 7]$. Because $A[i] = i$ for $i = 0 \ldots 4$, we compute and store the corresponding rows of $M$. Then we skip rows 5 and 6 because $A[5] = 3$ and $A[6] = 4$, but we consider row $p = 7$. To compute it we need to retrieve the row corresponding to $A[p - 1] = A[6] = 4$, that has been previously stored.

4. Applications

4.1. Simple counting to compute scores

As mentioned in the introduction, the computation of the frequency of all the words in a text can be used to compute scores that compare the actual and expected frequency of each word to perform pattern discovery based on over- or under-representation. In this scenario we do not need to store the actual positions of all the words, but only of any two consecutive words. This reduces the working space to $O(n)$. Indeed, in pattern discovery applications one could compute the scores first (using $O(n)$ working space and $O(1)$ storing space per word), and then search for the locations only of those words that are over- or under-represented.

Let us call $P_{cur}$ the array of size $n$ for the current position $i$, and $P_{old}$ the array of the previous position $i - 1$. While we compute $P_{cur}$ we count its entries with value $\leq k$ and store this value in an array $N_{occ}$ of size $n$, at position $i$. Then we overwrite $P_{old}$ with the values of $P_{cur}$, and increment the current position $i$. We sketch the modified algorithm in Fig. 4. Since we just keep two consecutive rows of the original table $M$ at a time, we will not get access to the values that were assigned directly from their symmetric. However, this is not a problem. We compute the values $P_{cur}[j], \forall j \geq i$ for every current position $i$. If $P_{cur}[j] \leq k$ we update both $N_{occ}[i]$ and $N_{occ}[j]$ because if $x(i, i + m - 1)$ has an instance with $v$ mismatches at position $j$, then $x(j, j + m - 1)$ has an instance with $v$ mismatches at position $i$.

4.2. Computing co-occurrences with mismatches

Our index provides a simple way to compute co-occurrences at, or within, distance $d$ between words of length $m$ that occur in the text at least once exactly. Suppose we are given two words $w_i = x(s_i, s_i + m - 1)$ with $i = 1, 2$, a distance $d$
and a number of allowed mismatches $k$. We are looking for segments $x_r \ldots x_r+d+2m-1$ such that:

(i) the Hamming distance between $w_1$ and $x_r \ldots x_r+m-1$ is $k$

(ii) the Hamming distance between $w_2$ and $x_r+m+d \ldots x_r+d+2m-1$ is $k$

We assume that building table $M$ we also fill the sets $\mathcal{L}(s, m)$. This does not change the time and space complexity since the sets $\mathcal{L}$ are just different a way to arrange the values of each row in $M$. We then consider the two set $\mathcal{L}(s_1, m)$ and $\mathcal{L}_k(s_2, m)$. If the former has the smallest cardinality, then we add to its members a value equal to $d + m$. Otherwise we subtract $d + m$ from the members of $\mathcal{L}_k(s_2, m)$. Finally we intersect the two sets. Since the sets are sorted we can use ad hoc fast intersection algorithms [5].

4.3. Approximate string matching

The table described in Section 3 can be used as an index to search the text for instances with mismatches of any word $w$ in $\Sigma^m$. If the word $w$ occurs in the text we simply look up the values in its corresponding row. Otherwise we sample a word $v$ from the text and use its indexed occurrences to find where $w$ occurs with $k$ mismatches. The efficiency of this algorithm depends on many parameters: (i) how we sample $v$; (ii) the Hamming distance between $w$ and $v$; (iii) the size of the sets $\mathcal{L}$ of $v$ that need to be examined; the source that generated the input sequence (random, biological, etc.). An assessment of the performance of this application requires an extended statistical analysis and comparison with indexed-based methods for the $k$-difference problem[7] that are currently under study and will be material for a separate paper.

Acknowledgements

The author would like to thank Prof. R. Baeza-Yates for posing the problem in terms of approximate string matching, and Prof. A. Apostolico for useful advises. Work by this author was supported in part by the Italian Ministry of University and Research under the Bi-National Project FIRB RBIN04BYZ7, and by the Research Program of the University of Padova.

References