Finding the Position of the $k$-Mismatch and Approximate Tandem Repeats

Haim Kaplan$^1$, Ely Porat$^2$, and Nira Shafrir$^1$

1 School of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel
{haimk, shafrirn}@post.tau.ac.il
2 Department of Mathematics and Computer Science, Bar-Ilan University, Ramat-Gan 52900, Israel
porately@cs.biu.ac.il

Abstract. Given a pattern $P$, a text $T$, and an integer $k$, we want to find for every position $j$ of $T$, the index of the $k$-mismatch of $P$ with the suffix of $T$ starting at position $j$. We give an algorithm that finds the exact index for each $j$, and algorithms that approximate it. We use these algorithms to get an efficient solution for an approximate version of the tandem repeats problem with $k$-mismatches.

1 Introduction

Let $P$ be a pattern of length $m$ and let $T$ be a text of length $n$. Let $T(i, \ell)$ denote the substring of $T$ of length $\ell$ starting at position $i$. In the $k$-mismatch problem we determine for every $1 \leq j \leq n - m + 1$, if $T(j, m)$ matches $P$ with at most $k$ mismatches. In case $T(j, m)$ does not match $P$ with at most $k$ mismatches we compute the position $k(j)$ in $P$ of the $k$-mismatch. In case $T(j, m)$ matches $P$ with at most $k$ mismatches we compute the position of the last mismatch if there is at least one mismatch.

Several classical results are related to the $k$-mismatch problem. Abrahamson [1], gave an algorithm that finds for each $1 \leq j \leq n - m + 1$, the number of mismatches between $T(j, m)$ and $P$. The running time of Abrahamson’s algorithm is $O(n\sqrt{m \log m})$. Amir et. al. [2], gave an algorithm that for each $1 \leq j \leq n - m + 1$, determines if the number of mismatches between $T(j, m)$ and $P$ is at most $k$. running time of this algorithm is $O(n\sqrt{k \log k})$. Both of these algorithms do not give any information regarding the position of the last mismatch or the position of the $k$-mismatch. This information is useful for applications that want to know not only if the pattern matches with at most $k$-mismatches, but also want to know how long is the prefix of the pattern that matches with at most $k$-mismatches.

The major technique used by the algorithms of Abrahamson and of Amir et. al. is convolution. Lets fix a particular character $x \in \Sigma$. Suppose we want to compute for every $1 \leq j \leq n - m + 1$, the number of places in which an $x$ in $P$ does not coincide with an $x$ in $T$ when we align $P$ with $T(j, m)$. We can

---

1 We always assume that $i \leq n - m + 1$ when we use this notation.
perform this task by computing a convolution of a binary vector \( P(x) \) of length \( m \), and a binary vector \( T(x) \) of length \( n \) as follows. The vector \( P(x) \) contains 1 in every position where \( P \) contains the character \( x \) and 0 in all other positions. The vector \( T(x) \) contains 1 in every position where \( T \) does not contain \( x \) and 0 in every position where \( T \) contains \( x \). We can perform the convolution between \( P(x) \) and \( T(x) \) in \( O(n \log m) \) time using the Fast Fourier Transform. So if \( P \) contains only \( |\Sigma| \) different characters we can count for each \( 1 \leq j \leq n - m + 1 \), the number of mismatches between \( T(j, m) \) and \( P \) in \( O(|\Sigma|n \log m) \). We do that by performing \( |\Sigma| \) convolutions as described above, one for each character in \( P \), and add up the mismatch counts.

There is a simple deterministic algorithm for the \( k \)-mismatch problem that runs in \( O(nk^{2/3} \log^{1/3} m) \) time and \( O(n) \) space of Landau and Vishkin [8]. They construct a suffix tree for the text and the pattern, with a data structure for lowest common ancestor (LCA) queries, to allow constant-time jumps over equal substrings in the text and pattern. The algorithm of Landau and Vishkin finds for each \( j \) the position of the \( k \)-mismatch (or the last mismatch if there are less than \( k \) mismatches) between \( T(j, m) \) and \( P \) in \( O(k) \) time. It does that by performing at most \( k \) LCA queries on the appropriate substrings of the text and the pattern. We give an alternative algorithm that runs in \( O(nk^{2/3} \log^{1/3} m \log k) \) time and linear space.

To see why the bound of \( O(nk^{2/3} \log^{1/3} m) \), may be natural, consider a pattern of length \( m = O(k) \). In this case, we can solve the problem using the method of Abrahamson [11]. We divide the pattern into \( k^{2/3}/\log^{1/3} k \) blocks, each block of size \( z = O(k^{2/3} \log^{1/3} k) \). By applying the algorithm of Abrahamson with the first block as the pattern, we determine in \( O(n\sqrt{z \log z}) = O(nk^{2/3} \log^{2/3} k) \) time, the number of mismatches of each text location with the first block. Similarly, by applying the method of Abrahamson to each of the subsequent \( k^{2/3}/\log^{1/3} k \) blocks of the pattern, and accumulating the number of mismatches for each text position, we know in \( O(nk^{2/3} \log^{1/3} k) \) time for each text position, which block contains the \( k \)-mismatch. Moreover we also know for each text position the number of mismatches in the blocks preceding the one that contains the \( k \)-mismatch. With this information, we can find for each text position the \( k \)-mismatch in the relevant block in \( O(k^{2/3} \log^{1/3} k) \) time by scanning the block character by character looking for the appropriate mismatch. It is not clear how to get a better bound even for this simple example.

We also define the approximate \( k \)-mismatch problem. This problem have an additional accuracy parameter \( \epsilon \). The task is to determine for every \( 1 \leq j \leq n - m + 1 \) a position \( k(j) \) in \( P \) such that the number of mismatches between \( T(j, k(j)) \) and \( P(1, k(j)) \) is at least \( (1 - \epsilon)k \) and at most \( (1 + \epsilon)k \), or report that there is no such position.

We give a deterministic and randomized algorithms for the approximate \( k \)-mismatch problem. We describe the deterministic algorithm in Section 3. The running time of this algorithm is \( O((n/3)\sqrt{k \log^3 m}) \). In Sect. 4 we give a randomized algorithm with running time of \( O(\frac{n}{\epsilon^2} \log n \log^3 m \log k) \). The randomized algorithm guarantees that for each \( j \) the number of mismatches between