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 ℓ starting at position i.[3] 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

[3] 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 contains 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 $O(nk)$ 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 $O(nk^{\frac{2}{3}} \log^{1/3} m \log k)$ time and linear space.

To see why the bound of $O(nk^{\frac{2}{3}} \log^{1/3} m)$, may be natural, consider a pattern of length $m = O(k)$. In this case, we can solve the problem using the method of Abrahamson [1]. We divide the pattern into $k^{\frac{1}{3}} / \log^{1/3} k$ blocks, each block of size $z = O(k^{\frac{2}{3}} \log^{1/3} k)$. By applying the algorithm of Abrahamson with the first block as the pattern, we determine in $O(n\sqrt{z \log z}) = O(nk^{\frac{1}{3}} \log^{2/3} k)$ 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 $k^{\frac{1}{3}} / \log^{1/3} k$ blocks of the pattern, and accumulating the number of mismatches for each text position, we know in $O(nk^{\frac{2}{3}} \log^{1/3} k)$ 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 $O(k^{\frac{2}{3}} \log^{1/3} k)$ 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 ϵ. 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 $O((n/\epsilon^3)\sqrt{k} \log^3 m)$. 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

$T(j,k(j))$ and $P(1,k(j))$ is at least $(1-\epsilon)k$ and at most $(1+\epsilon)k$ with high probability.[4]

A position $k(j)$ computed by our algorithms for the *approximate k-mismatch problem* may not contain an actual mismatch. That is, the character $k(j)$ of P may in fact be the same as character $j+k(j)-1$ of T. We can change both algorithms such that $k(j)$ would always be a position of a mismatch in O(n) time as follows. For a string S we denote by S^R the string obtained by reversing S. We build a suffix tree for T^R and P^R, with a data structure for lowest common ancestor (LCA) queries in constant time. For each position j in T we perform an LCA query for the suffixes $(P(1,k(j)))^R$ of P^R and $(T(1,j+k(j)-1))^R$ of T^R. Let h be the string depth of the resulting node. Clearly h is the length of the longest common prefix of $(P(1,k(j)))^R$ and $(T(1,j+k(j)-1))^R$, and $k(j)-h$ is the position of the last mismatch between P and $T(j,m)$ prior to position $k(j)$. We change $k(j)$ to $k(j)-h$.

In Sect. 5, we use our algorithms for the k-mismatch problem to solve an approximate version of the k-mismatch tandem repeats problem. The *exact tandem repeats problem* is defined as follows. Given a string S of length n, find all substrings of S of the form uu. Main and Lorentz [9] gave an algorithm that solves this problem in $O(n\log n+z)$ time, where z is the number of tandem repeats in S. Repeats occur frequently in biological sequences, but they are usually not exact. Therefore algorithms for finding approximate tandem repeats were developed. The *k-mismatch tandem repeats problem* is defined as follows. Given a string S and a parameter k find all substrings uv of S such that $|u|=|v|>k$ and the number of mismatches between u and v is at most k. The best known algorithm for this problem is due to Landau, Schmidt and Sokol [7] and it runs in $O(nk\log(n/k)+z)$ time, where z is the number of k-mismatch tandem repeats.

We define the *approximate k-mismatch tandem repeats problem* which is a relaxation of the k-mismatch tandem repeats problem. In this relaxation we require that the algorithm will find all substrings uv of S such that $|u|=|v|>k$ and the number of mismatches between u and v is at most k, but we also allow the algorithm to report substrings uv such that the number of mismatches between u and v is at most $(1+\epsilon)k$. Using our algorithm for the k-mismatch problem we get an algorithm for approximate k-mismatch tandem repeats that runs in $O((n/\epsilon)k^{\frac{2}{3}}\log^{1/3}n\log k\log(n/k)+z)$ time. Using our deterministic algorithm for the approximate k-mismatch problem we get an algorithm for approximate k-mismatch tandem repeats that runs in $O((n/\epsilon^4)\sqrt{k}\log^3 n\log(n/k)+z)$ time. We can also use the randomized algorithm of Sect. 4 and get an algorithm that reports all k-mismatch tandem repeats with high probability, and possibly tandem repeats with up to $(1+\epsilon)k$ mismatches in $O(\frac{n}{\epsilon^3}\log^3 n\log k\log(n/k)+z)$ time.

Preliminaries: A string s is *periodic* with period u, if $s=u^jw$, where $j\geq 2$ and w is a prefix of u. The *period* of s is the shortest substring u such that $s=u^jw$ and w is a prefix of u.

A *break* of s is an aperiodic substring of s. An *ℓ-break* is a break of length ℓ. We choose a parameter $\ell<k$ (the value of ℓ will be decided later). We use

[4] By high probability we mean probability that is polynomially small in n.

the method of [3] to find a partition of the pattern into ℓ-breaks separated by substrings shorter than ℓ, or periodic substrings with period of length at most $\ell/2$. We call the substrings that separate the breaks *periodic stretches.*

In Sect. 2, we show how to solve the k-mismatch problem when the pattern P contains at most $2k$ ℓ-breaks in the time and space bounds mentioned above. In case the pattern P contains more than $2k$ ℓ-breaks, we reduce it to the case where P contains $2k$ ℓ-breaks as follows.

Assume P contains more than $2k$ ℓ-breaks and let P' be the prefix of P with exactly $2k$ ℓ-breaks. We run our algorithm using P' rather than P. Our algorithm also finds all positions in T that match P' with at most k mismatches. Amir et. al. [2] proved that at most n/ℓ positions of the text T match P' with at most k mismatches. After running our algorithm and finding these positions we use the algorithm of Landau and Vishkin [8] to check whether each of these positions matches the original pattern P with at most k mismatches, and to find the location of the k-mismatch in case it does not. The total time it takes to check all of these positions is $O(nk/\ell)$. Therefore we assume from now on that the pattern P contains at most $2k$ ℓ-breaks, and that the running time of our algorithm is $\Omega(nk/\ell)$.

2 Finding the position of the k-mismatch

We describe an algorithm that solves the problem in $O(nk^{\frac{3}{4}} \log^{1/4} m)$ time and $O(n)$ space. In the full version of this paper we show how to add another level of recursion to this algorithm and get an algorithm whose running time is $O(nk^{\frac{2}{3}} \log^{1/3} m \log k)$ and uses $O(n)$ space.

Recall that we assume that the pattern contains $O(k)$ breaks, which are substrings of length at most ℓ, and at most $2k$ periodic stretches. Let A be a periodic stretch let x be its period, $|x| \leq \ell/2$. Let x' be the lexicographically first cyclic rotation of x. We call x' the *canonical period* of A. We can write $A = yx'^i z, i \geq 0$, where y is a prefix of x, (y may be empty), and z is a prefix of x' which may be empty. Let $A' = x'^i$. We add y and z to the set of breaks. We redefine the term *break* to include also the above substrings. The string A' is the new periodic stretch. We added to the set of breaks a total of $O(k)$ substrings each of length at most ℓ. After this preprocessing, the set of all different periods of the periodic stretches of the pattern contains only canonical periods, and thus it doesn't contain two periods that are cyclic rotations one of the other. In addition, all periodic stretches with period u are of the form $u^i, i > 0$.

Choosing a prefix of the pattern: We now show how to choose a prefix S of the pattern for which we can find the position of the k-mismatch with $T(j, |S|)$ or determine that S matches $T(j, |S|)$ with less than k-mismatches. We also prove that S cannot match $T(j, |S|)$ with at most k-mismatches in too many positions j. We assume that P contains $O(k)$ breaks, which are substrings of length at most ℓ, and at most $2k$ periodic stretches. All periodic stretches are of the form u^i, where u is a canonical period. We partition each periodic stretch into segments of length ℓ. We ignore the segments that are not fully contained in a periodic stretch.

Let S be the shortest prefix of P that satisfies at least one of the following criteria, or P itself if no prefix of P satisfies at least one of these criteria.

1. S contains a multiset A of $2k$ segments of periodic stretches, such that at most k/ℓ are of the same canonical period.
2. S contains a multiset of $2k$ characters in which each character appears at most k/ℓ times.

We use the following definitions. Let C be the set of canonical periods of the periodic stretches in S. We define a period $u \in C$ to be to *frequent* in S, if there are more than k/ℓ segments in the above partition with period u and *rare* otherwise. Similarly, we define a character to be *frequent* in S, if it appears more than k/ℓ times in S, and *rare* otherwise. The prefix S has the following properties.

1. C contains at most 2ℓ frequent periods. If C contains more than 2ℓ frequent periods, then we can obtain a shorter S satisfying (1) by taking the shortest prefix that contains k/ℓ segments of each of exactly 2ℓ frequent periods. By a similar argument, the total number of segments of periodic stretches that belong to rare periods in S is at most $2k$.
2. S contains at most 2ℓ frequent characters. Furthermore, the total number of occurrences of rare characters in S is at most $2k$.

We add to the set of breaks all rare periodic stretches. By property 1 we added $O(k)$ breaks of length at most ℓ. Following these changes, S contains $O(k)$ breaks. The set C of periods of the periodic stretches is of size $O(\ell)$.

Finding the position of the k-mismatch in S: Next we show how to find the position of the k-mismatch of each location of the text T with a prefix S of the pattern chosen as in Sect. 2. Recall that S contains $O(k)$ breaks and at most $2k$ periodic stretches, and satisfies Properties 1 and 2.

We partition the pattern into at most $O(k/y)$ *substrings* each contains at most y breaks, at most y rare characters and at most y periodic stretches. First we compute for each text position j the substring $W(j)$ of P that contains the k-mismatch of P with $T(j, m)$, or determine that P matches $T(j, m)$ with less than k-mismatches.

To do that we process the substrings sequentially from left to right, maintaining for each text position j the cumulative number of mismatches of the text starting at position j with the substrings processed so far. We denote this cumulative mismatch count of position j by $r(j)$. Let the next substring W of P that we process start at position i of the pattern. For each text position j, we compute the number of mismatches of $T(j, |W|)$ with W and denote it by $c(j)$. (We show below how to do that.) Then, for each text position j for which we haven't yet found the substring that contains the k-mismatch, we update the information as follows. If $r(j) + c(j + i) < k$, we set $r(j) = r(j) + c(j + i)$. Otherwise, $r(j) + c(j + i) \geq k$, and we set $W(j)$ to be W.

We now show how to find the number of mismatches between a substring W of S and $T(j, |W|)$ for every $1 \leq j \leq n - |W| + 1$. We do that by separately counting the number of mismatches between occurrences of frequent characters in W and the corresponding characters of $T(j, |W|)$, and the number of mismatches between occurrences of rare characters in W and the corresponding characters of $T(j, |W|)$. Then we add these two counts.

By Property 2, W contains at most 2ℓ frequent characters. For each frequent character x we find the number of mismatches of the occurrences of x in W with the corresponding characters in $T(j, |W|)$ for all j, by performing a convolution as described in the introduction. We perform $O(\ell)$ convolutions for each of the $O(k/y)$ substrings, so the total time to perform all convolutions is $O((k/y)\ell n \log m)$.

It remains to find the number of mismatches of rare characters in W with the corresponding characters in $T(j, |W|)$. We do that using the algorithm of Amir et. al. [2]. This algorithm counts the number of mismatches of a pattern which may contain don't care symbols with each text position. The running time of this algorithm is $O(n\sqrt{g \log m})$, where g is the number of characters in the pattern that are not don't cares. We run this algorithm with a pattern which we obtain from W by replacing each occurrence of a frequent character by a don't care symbol, and the text T. We obtain for each j the number of mismatches between rare characters in W and the corresponding characters in $T(j, |W|)$. Since W contains at most y rare characters, the running time of this application of the algorithm of Abrahamson is $O(n\sqrt{y \log m})$. So for all $O(k/y)$ substrings this takes $O((k/y)n\sqrt{y \log m}) = O(n(k/y^{1/2})\sqrt{\log m})$ time.

We now show how to find the position of the k-mismatch within the substring $W(j)$ that contains it for each text position j. We assume that each substring contains y breaks and y periodic stretches. Each periodic stretch is of the form u^i, where $u \in C$, and $|C| \leq 2\ell$.

We begin by finding for each text position which periodic stretch or break contains the k-mismatch. We find it by performing a binary search on the periodic stretches and breaks in $W(j)$. We do the binary search simultaneously for all text positions j. After iteration h of the binary search, for each text position we focus on an interval of $y/2^h$ consecutive breaks and periodic stretches in $W(j)$ that contain the k-mismatch between $W(j)$ and the corresponding substring of $T(j, m)$. In particular after $\log y$ iterations, we know for each text position which periodic stretch or break contains the k-mismatch.

At the first iteration of the binary search we compute the number of mismatches in the first $y/2$ of the periodic stretches and breaks of $W(j)$. From this number we know if the k-mismatch is in the first $y/2$ breaks and periodic stretches or in the last $y/2$ breaks and periodic stretches of $W(j)$. In iteration h, let $I(j)$ be the interval of $y/2^h$ consecutive breaks and periodic stretches in $W(j)$ that contains the k-mismatch between $W(j)$ and the corresponding piece of $T(j, m)$. We compute the number of mismatches between the first $y/2^{h+1}$ breaks and periodic stretches in $I(j)$ and the corresponding part of $T(j, m)$. Using this count we know if to proceed with the first half of $I(j)$ or the second half of $I(j)$.

We describe the first iteration of the binary search. Subsequent iterations are similar. We count the number of mismatches in each of the first $y/2$ breaks in $W(j)$ and $T(j,m)$ by comparing them character by character in $y\ell/2$ time for a specific j, and $ny\ell/2$ total time. To count the number of mismatches in each of the first $y/2$ periodic stretches we process the different periods in C one by one. For each period $u \in C$ and each text position j we count the number of mismatches in periodic stretches of u among the first $y/2$ periodic stretches of $W(j)$. The sum of these mismatch counts over all periods $u \in C$ gives us the total number of mismatches in the first $y/2$ periodic stretches of $W(j)$ and $T(j,m)$ for every text position j.

Let $u \in C$. We compute the number of mismatches of u with each text location using the algorithm of Abrahamson [1] in $O(n\sqrt{\ell \log \ell})$ time. We build a data structure that consists of $|u|$ prefix sums arrays $A_i, i = 1, \cdots, |u|$, each of size $n/|u|$. We use these arrays to find the number of mismatches of periodic stretches of u among the first $y/2$ periodic stretches of $W(j)$ for all text positions j. The total size of the arrays is $O(n)$.

The entries of array A_i correspond to the text characters at positions β such that β modulo $|u| = i$ modulo $|u|$. The first entry of array A_i contains the number of mismatches between $T(i,|u|)$ to u that was computed by the algorithm of Abrahamson. Entry j in A_i contains the number of mismatches between $T(i, j|u|)$ and u^j. It is easy to see that based on entry $j-1$, entry j in A_i can be computed in $O(1)$ time. Suppose we need to find the number of mismatches of $T(i + j|u|, r|u|)$ with a periodic stretch u^r. The number of mismatches can be computed in $O(1)$ time given A_i. If $j = 0$, then the number of mismatches is $A_i[r]$. If $j > 0$, then the number of mismatches is $A_i[j+r] - A_i[j]$.

In each iteration of the binary search we repeat the procedure above for every $u \in C$. Since $|C| = O(\ell)$ we compute the number of mismatches of all periodic stretches in the first $y/2$ periodic stretches of $W(j)$ for all j, in $O(n\ell^{3/2}\sqrt{\log \ell})$ time. Summing up over all iterations the time of counting the number of mismatches within breaks and the time of counting the number of mismatches within periodic stretches, we obtain that the binary search takes $O(n\ell^{3/2}\sqrt{\log \ell}\log y) + O(ny\ell)$ time.

We now know for each text position which periodic stretch or break contains the position of the k-mismatch. If the k-mismatch is contained within a break we find it in $O(\ell)$ time by scanning the break character by character. If the k-mismatch is contained in a periodic stretch, then we find it as follows. For each $u \in C$ we build the $n/|u|$ prefix sum arrays A_i, as described above. We then compute the position of the k-mismatch, for all text position for which the k-mismatch occurs with a periodic stretch of period u. Given such text position, we perform a binary search on the appropriate prefix sum array to locate a segment of length $|u|$ within the periodic stretch that contains the k-mismatch. The binary search is performed on a sub-array of length at most $m/|u|$ in $O(\log m)$ time. At the end of the binary search, we found the segment of length $|u| < \ell$ that contains the k-mismatch, we search in this segment sequentially in $O(\ell)$ time to find the k-mismatch. We repeat this process for all the periods in C.

Summing over all stages we obtain that the total running time of the algorithm is $O((k/y)n\ell\log m)+O(n(k/y^{1/2})\sqrt{\log m})+O(n\ell^{3/2}\sqrt{\log\ell}\log y)+O(ny\ell)$. The space used by the algorithm is $O(n)$.

To complete the analysis we prove in the full version of this paper that if S is not equal to P, then T contains at most n/ℓ positions that match S with at most k mismatches. In these cases we use the algorithm of Landau and Vishkin [8] to find the position of the k-mismatch (or the last mismatch if there are less than k-mismatches) of each of these positions with the pattern in $O(nk/\ell)$ time. We also recall that we have to take into account the overhead of $O(nk/\ell)$ time of the reduction in Sect. 1 to a pattern with at most $O(k)$ breaks and periodic stretches.

So if we add the extra $O(nk/\ell)$ overhead to the overall running time and choose ℓ and y to balance the expressions (and thereby minimize the running time) we get that $\ell = k^{1/4}/\log^{1/4} m, y = \sqrt{k\log m}$ and a running time of $O(nk^{3/4}\log^{1/4} m)$.

3 Approximate k-mismatch

In this section we sketch how to obtain an algorithm for the approximate k-mismatch problem whose running time is $O(n(1/\epsilon^3)\sqrt{k}\log^3 m)$. The algorithm is similar to the algorithm of Sect. 2. The main difference is that instead of using convolutions or the algorithm of Abrahamson [1] (that uses convolutions), to count the number of mismatches of various parts of the pattern and the text, we use the algorithm of Karloff [6]. Given a pattern P and a text T, the algorithm of Karloff [6], finds for every text position $1 \le j \le n-m+1$, a number $g(j)$ such that $m(j) \le g(j) \le (1+\epsilon)m(j)$, where $m(j)$ is the exact number of mismatches between P and $T(j,m)$.

We choose a prefix S to satisfy the first of the two criteria of Sect. 2. We partition S into $O((1/\epsilon)k/y)$ substrings each containing at most ϵy breaks and at most ϵy periodic stretches. We use the algorithm of Karloff [6] to approximately count the number of mismatches of each text position and each substring of P in $O(n/\epsilon^2\log^3 m)$ time. Then we know for each j which substring of P contains the k-mismatch with $T(j,m)$. We then search within the substring by a binary search as in Section 2. Here we set $\ell = \sqrt{k}/\log k$, and $y = \sqrt{k}$, so $y\ell = k/\log k$, and therefore the total length of the breaks within each substring is at most $\epsilon y\ell = \epsilon k/\log k$. This allows us to ignore the breaks when looking for the position within a substring.

4 A Randomized Algorithm for approximate k–mismatch

We assume w.l.o.g. that the alphabet Σ consists of the integers $\{1,\cdots,|\Sigma|\}$. The algorithm computes signatures for substrings of the pattern and the text. These signatures are designed such that from the signatures of two strings we can quickly approximate the number of mismatches between the two strings. We construct a random string R *of sparsity* k by setting $R[i]$ to 0 with probability

$(1-\frac{1}{k})$, and setting $R[i]$ to be a random integer with probability $\frac{1}{k}$, for every $i = 1, \cdots, |R|$. We choose a random integer from a space Π of size polynomial in n. For a string W and a random string R with sparsity k, we define the signature of W with respect to R as $Sig_k(W, R) = \sum_{i=1}^{|W|} W[i]R[i]$.

Let W_1 and W_2 be two strings of the same length. If W_1 and W_2 agree in all positions where $R[i] \neq 0$, then $Sig_k(W_1, R) = Sig_k(W_2, R)$. On the other hand, if W_1 and W_2 disagree in at least one position i where $R[i] \neq 0$, then $Sig_k(W_1, R) = Sig_k(W_2, R)$ with probability at most $\frac{1}{|\Pi|}$. Let us call the latter event a *bad event*. Our algorithm compares sub-quadratic number of signatures so by choosing Π large enough, we can make the probability that a bad event ever happens polynomially small. Therefore, we assume in the rest of the section that such event does not happen.

For $k \geq 2$ we define an algorithm A_k as follows. The input to A_k consists of a substring S of T and a substring W of P such that S and W are of the same length. Let y be the true number of mismatches between S and W. The algorithm A_k either detects that $y > 2k$, or detects that $y < k$, or returns an estimate y' of y. The algorithm A_k works as follows. Let $q = \frac{c}{\epsilon^2} \log n$ for some large enough constant c that we determine later, and let $b = |W| = |S|$. Algorithm A_k takes q random strings $R_1, \cdots, R_q$ of length b and sparsity k and compares $Sig_k(W, R_i)$ and $Sig_k(S, R_i)$ for $i = 1, \ldots, q$. Let z be the number of equal pairs of signatures. If $z \geq (1-\epsilon)q(1-\frac{1}{k})^{k/2}$ then A_k reports that the number of mismatches between S and W is smaller than k. If $z \leq (1+\epsilon)q(1-\frac{1}{k})^{3k}$ then A_k reports that the number of mismatches between S and W is greater than $2k$. Otherwise let y' be the largest integer such that $z \leq q(1-\frac{1}{k})^{y'}$. We then return y' as our estimate of y.

Using standard Chernoff bounds we establish that A_k satisfies the following properties with high probability. (Proof omitted from this abstract.)

1. If $y \leq k/2$ then A_k reports that the number of mismatches is smaller than k.
2. If $y \geq 3k$ then A_k reports that the number of mismatches is larger than $2k$.
3. If $k \leq y \leq 2k$ then A_k gives an estimate y' to y.
4. Whenever A_k gives an estimate y' of y then $(1-\epsilon)y \leq y' \leq (1+\epsilon)y$. (This can happen if $k/2 < y < 3k$ and happens with high probability if $k \leq y \leq 2k$.)

For $k < 2$ we build a generalized suffix tree for P and T. We use this suffix tree to check whether the number of mismatches between a substring of P and a substring of T is at most 2, and if so to find it exactly, by the method of Landau and Vishkin. We shall refer to this procedure as A_0.

We are now ready to describe the algorithm. To simplify the presentation, we assume that k is a power of 2. Our algorithm compares substrings of P and T, by comparing their signatures using the algorithm A_j, for some $j \leq k$ which is a power of two, and we always compare substrings of length which is a power of two. We prepare all signatures required by for these applications of A_j in a preprocessing phase using convolutions as follows.

For any 2^j, $0 \leq j \leq \lfloor \log m \rfloor$, and for any 2^i, $0 \leq i \leq \log k$, we generate independently at random $q = \frac{c}{\epsilon^2} \log n$ strings $R_1, \cdots, R_q$, of sparsity 2^i and length 2^j. For each random string R_l of length 2^j, we compute the signature of every substring of T of length 2^j with R_l by a convolution of T and R_l. We compute the signature of every substring of P of length 2^j with R_l by a convolution of P and R_l. We compute a total of $\frac{c}{\epsilon^2} \log n \log m \log k$ signatures in $O(\frac{n}{\epsilon^2} \log n \log^2 m \log k)$ time.

We find the approximated location of the k-mismatch of $T(j, m)$ with P by a binary search as follows. To simplify the presentation we assume that m is a power of 2 and we show in the full version of the paper how to handle patterns whose length is not a power of 2. We compute the approximate number of mismatches y', between $P(1, m/2)$ and $T(j, m/2)$. We find y' by performing a binary search on $A_j(P(1, m/2), T(j, m/2))$, for $j = 0, 2, 4, \cdots, k$. We first apply $A_{\sqrt{k}}(P(1, m/2), T(j, m/2))$, if $A_{\sqrt{k}}$ reports that the number of mismatches is smaller than $\sqrt{k}/2$ we repeat the process for $j = 0, 2, 4, \cdots \sqrt{k}/2$. If $A_{\sqrt{k}}$ reports that the number of mismatches is larger than $2\sqrt{k}$, we repeat the process for $j = 2\sqrt{k}, \cdots, k$. Otherwise the algorithm gave us a good estimation y' of the number of mismatches between $P(1, m/2)$, and $T(j, m/2)$. Once we find y' we proceed as follows. If $y' > (1 + \epsilon)k$ we search recursively for the position of the k-mismatch in $P(1, m/2)$. If $y' < (1 - \epsilon)k$ we search recursively for the $k - y'$-mismatch in $P(m/2 + 1, m/2)$. If $(1 - \epsilon)k \leq y' \leq (1 + \epsilon)k$, the approximated k-mismatch is at position $m/2$ of the pattern and we are done.

It is easy to see that the running time of the search is $O(\frac{n}{\epsilon^2} \log n \log m \log logk)$. The total running time of the algorithm is $O(\frac{n}{\epsilon^2} \log n \log^2 m \log k)$.

5 Approximate Tandem Repeats

We first describe the algorithm for exact tandem repeats. Then we describe the algorithm for the k-mismatch tandem repeats that runs in $O(nk \log(n/k) + z)$. Finally we show how to change this algorithm to get our algorithm. Let S be the input string of length n. Let $S[i \cdots j]$ be the substring of S that starts at position i and ends at position j, and recall that $S[i \cdots j]^R$ is the string obtained by reversing $S[i \cdots j]$. Let $S[i]$ be the character at position i.

We now describe the exact algorithm of Main and Lorentz [9]. Let $h = \lfloor n/2 \rfloor$. Let $u = S[1 \cdots h]$ be the first half of S, and let $v = S[h + 1 \cdots n]$ be the second half of S. The algorithm finds all tandem repeats that contain $S[h]$ and $S[h+1]$. That is repeats that are not fully contained in u and are not fully contained in v, and then calls itself recursively on u to find all tandem repeats contained in the first half of S, and calls itself recursively on v to find all tandem repeats contained in the second half of S.

The repeats that contain $S[h]$ and $S[h+1]$ are classified into *left repeats* and *right repeats*. *Left repeats* are all tandem repeats zz where the first copy of z contains h. *Right repeats* are all tandem repeats zz where the second copy of z contains h. We describe how to find all left repeats. Right repeats are found similarly. We build a suffix tree that supports LCA queries in $O(1)$ time for S and S^R. The algorithm for finding left repeats in S has $n/2$ iterations. In the i-th iteration, we find all left repeats of length $2i$ as follows.

1. Let $j = h + i$.
2. Find the longest common prefix of $S[h \cdots n]$ and of $S[j \cdots n]$. Let ℓ_1 be the length of this prefix.
3. Find the longest common prefix of $S[1 \cdots h-1]^R$ and of $S[1 \cdots j-1]^R$. Let ℓ_2 be the length of this prefix.
4. If $\ell_1 + \ell_2 \geq i$ there is at least one tandem repeat of length $2i$. All left repeats of length $2i$, begin at positions $\max(h-\ell_2, h-i+1), \cdots, \min(h+\ell_1-i, h)$.

Using the suffix tree we can find each longest common prefix in $O(1)$ time. Therefore, we can find an implicit representation of all left repeats of length $2i$ in $O(1)$ time. The total time it takes to find all left and right repeats for $h = \lfloor n/2 \rfloor$ is $O(n)$, and the total running time of the algorithm is $O(n \log n + z)$.

The algorithm of [7] for finding k-mismatch tandem repeats is an extension of the algorithm of Main and Lorentz [9]. Here we stop the recursion when the length of the string is at most $2k$, and in each iteration we compute only repeats of length greater than $2k$. Given $h = \lfloor n/2 \rfloor$ and $i > k$ the algorithm for finding all k-mismatch left repeats of size $2i$ is as follows.

1. Let $j = h + i$.
2. We find the positions of the first $k+1$ mismatches of $S[h \cdots n]$ and $S[j \cdots n]$ by performing $k+1$ successive LCA queries on the suffix tree of S. Let ℓ_1 be the position of the $(k+1)$-mismatch of the two strings.
3. Similarly, we find the positions of the first $k+1$ mismatches of $S[1 \cdots h-1]^R$ and $S[1 \cdots j-1]^R$ by performing $k+1$ successive LCA queries on a suffix tree of S^R. Let ℓ_2 be the position of the $(k+1)$-mismatch of the two strings.
4. If $\ell_1 + \ell_2 \geq i$, the k-mismatch tandem repeats will be those at positions $\max(h-\ell_2, h-i+1) \cdots \min(h+\ell_1-i, h)$ that have at most k mismatches. We can find all these positions in $O(k)$ time by merging the sorted list of item 2 containing the positions of the mismatches that are in $[h \cdots h+i]$ with the sorted list of item 3 containing the positions of the mismatches that are in $[h \cdots h+i]$. All positions in a segment between two successive elements in the merged list either all correspond to tandem repeats or none does. (See [7, 5] for more details).

The time it takes to find all left and right k-mismatch tandem repeats for $h = \lfloor n/2 \rfloor$ is $O(nk)$, and the total running time of the algorithm is $O(nk \log(n/k) + z)$.

We are now ready to describe our approximate tandem repeats algorithm for ϵ and k. We use the algorithm of Sect. 2 (with minor modifications and with different scaling of ϵ we can also use the algorithms of Sect. 3, and Sect. 4 instead). The algorithm has the same steps as the algorithm of [7]. The only difference is in the way left (and right) tandem repeats are computed. Let $h = \lfloor n/2 \rfloor$. Let the string $P_h = S[h \cdots n]$ and let $T_h = S[h \cdots n]\$^{n/2}$ be the string which is the catenation of P_h and the string $\$^{n/2}$, where \$ is a new character that doesn't appear in S. The string $\$^{n/2}$ is used to make sure that the text is always longer than the pattern, we ignore mismatches that are caused by it. Let $P_{h-1}^R = S[1 \cdots h-1]^R$ and let $T^R = S[1 \cdots n]^R$. We compute the left repeats as follows.

1. Compute the position of the i-mismatch between the text T_h and the pattern P_h, for $i = \epsilon k, 2\epsilon k, \cdots, k - \epsilon k, k$. We do that by running the algorithm of Sect. 2 once for every $i = \epsilon k, 2\epsilon k, \cdots, k - \epsilon k, k$. Let B_i be the vector that contains these positions. That is $B_i[r], r \geq h$ contains the position of the i-mismatch between $S[r \cdots n]$ and $S[h \cdots n]$.
2. Compute the position of the i-mismatch between the text T^R and the pattern P^R_{h-1}, for $i = \epsilon k, 2\epsilon k, \cdots, k - \epsilon k, k$ with the algorithm of Sect. 2. Let $B^R_i, i \in \{\epsilon k, 2\epsilon k, \cdots, k\}$ be the vector that contains these positions. That is $B^R_i[r], r \geq h$ contains the position of the i-mismatch between $S[1 \cdots r]^R$ and $S[1 \cdots h-1]^R$.
3. For each $r > k$ we find all approximate tandem repeats of length $2r$ whose first half contains h as follows. The q^{th} element in the sequence $B_{\epsilon k}[h + r], \cdots, B_k[h+r]$ contains the position of the $q\epsilon k$-mismatch between $S[h \cdots n]$ and $S[h+r \cdots n]$. The q^{th} element in the sequence $B^R_{\epsilon k}[h+r-1], \cdots, B^R_k[h+r-1]$ contains the position of the $q\epsilon k$-mismatch between $S[1 \cdots h-1]^R$ and $S[1 \cdots h+r-1]^R$. We activate the procedure of [7] that we described in item 4 of the previous algorithm, on these sequences of $O(1/\epsilon)$ positions of mismatches in $O(1/\epsilon)$ time. It is easy to see that this algorithm produces all tandem repeats with at most k mismatches. The algorithm may also report tandem reports with at most $(1 + 2\epsilon)k$-mismatches.

Items 1 and 2 that take $O((1/\epsilon)nk^{2/3} \log^{1/3} n \log k)$ time dominated the running time of each recursive call.
Therefore the total time is $O((1/\epsilon)nk^{2/3} \log^{1/3} n \log k \log(n/k) + z)$.

Acknowledgements: We thank Uri Zwick for suggesting to use prefix sum arrays in Sect. 2.

References

1. Karl Abrahamson. Generalized string matching. *SIAM J. Comput.*, 16(6):1039–1051, 1987.
2. Amihood Amir, Moshe Lewenstein, and Ely Porat. Faster algorithms for string matching with k mismatches. *J. Algorithms*, 50(2):257–275, 2004.
3. Richard Cole and Ramesh Hariharan. Approximate string matching: A simpler faster algorithm. *SIAM J. Comput.*, 31(6):1761–1782, 2002.
4. M. Crochemore and W. Rytter. *Text Algorithms.* Oxford Univ. Press, New-York, 1994. pp. 27-31.
5. Dan Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology.* Cambridge Univ. Press, 1997.
6. Howard J. Karloff. Fast algorithms for approximately counting mismatches. *Inf. Process. Lett.*, 48(2):53–60, 1993.
7. Gad M. Landau, Jeanette P. Schmidt, and Dina Sokol. An algorithm for approximate tandem repeats. *Journal of Computational Biology*, 8(1):1–18, 2001.
8. G.M. Landau and U. Vishkin. Efficient string matching in the presence of errors. In *Proc. 26th IEEE Symposium on Foundations of Computer Science*, pages 126–136, Los Alamitos CA, USA, 1985. IEEE Computer Society.
9. Michael G. Main and Richard J. Lorentz. An o(n log n) algorithm for finding all repetitions in a string. *J. Algorithms*, 5(3):422–432, 1984.