



## Faster algorithms for string matching with $k$ mismatches

Amihod Amir,<sup>a,b,1</sup> Moshe Lewenstein,<sup>a,\*</sup> and Ely Porat<sup>a</sup>

<sup>a</sup> Department of Mathematics and Computer Science, Bar-Ilan University, 52900 Ramat-Gan, Israel

<sup>b</sup> College of Computing, Georgia Institute of Technology, Atlanta, GA 30332-0280, USA

Received 1 July 2000

---

### Abstract

The *string matching with mismatches* problem is that of finding the number of mismatches between a pattern  $P$  of length  $m$  and every length  $m$  substring of the text  $T$ . Currently, the fastest algorithms for this problem are the following. The Galil–Giancarlo algorithm finds all locations where the pattern has at most  $k$  errors (where  $k$  is part of the input) in time  $O(nk)$ . The Abrahamson algorithm finds the number of mismatches at every location in time  $O(n\sqrt{m \log m})$ . We present an algorithm that is faster than both. Our algorithm finds all locations where the pattern has at most  $k$  errors in time  $O(n\sqrt{k \log k})$ . We also show an algorithm that solves the above problem in time  $O((n + (nk^3)/m) \log k)$ .

© 2003 Elsevier Inc. All rights reserved.

**Keywords:** Design and analysis of algorithms; Combinatorial algorithms on words; Approximate string matching; Hamming distance

---

### 1. Introduction

String matching, the problem of finding all occurrences of a given pattern in a given text, is a classical problem in computer science. The problem has pleasing theoretical features and a number of direct applications to “real world” problems. The Boyer–Moore [8] algorithm is directly implemented in the emacs “s” and *UNIX* “grep” commands.

---

\* Corresponding author.

*E-mail addresses:* [amir@cs.biu.ac.il](mailto:amir@cs.biu.ac.il) (A. Amir), [moshe@cs.biu.ac.il](mailto:moshe@cs.biu.ac.il) (M. Lewenstein), [porately@cs.biu.ac.il](mailto:porately@cs.biu.ac.il) (E. Porat).

<sup>1</sup> Partially supported by NSF grant CCR-96-10170, BSF grant 96-00509, and a BIU internal research grant.

Advances in multimedia, digital libraries, and computational biology have shown that a much more generalized theoretical basis of string matching could be of tremendous benefit [23,24]. To this end, string matching has had to adapt itself to increasingly broader definitions of “matching.” Two types of problems need to be addressed—*generalized matching* and *approximate matching*. In generalized matching, one still seeks all exact occurrences of the pattern in the text, but the “matching” relation is defined differently. The output is all locations in the text where the pattern “matches” under the new definition of match. The different applications define the matching relation. Examples can be seen in Baker’s *parameterized matching* [6] or Amir and Farach’s *less-than matching* [5]. The second model, and the one we are concerned with in this paper, is that of approximate matching. In approximate matching, one defines a distance metric between the objects (e.g., strings, matrices) and seeks all text location where the pattern matches the text by a pre-specified “small” distance.

One of the earliest and most natural metrics is the *hamming distance*, where the distance between two strings is the number of mismatching characters. Levenshtein [20] identified three types of errors, mismatches, insertions, and deletions. These operations are traditionally used to define the *edit distance* between two strings. The edit distance is the *minimum* number of edit operations one needs to perform on the pattern in order to achieve an exact match at the given text location. Lowrance and Wagner [21,27] added the *swap* operation to the set of operations defining the distance metric. Much of the recent research in string matching concerns itself with understanding the inherent “hardness” of the various distance metrics, by seeking upper and lower bounds for string matching under these conditions.

Let  $n$  be the text length and  $m$  the pattern length. Lowrance and Wagner proposed an  $O(nm)$  dynamic programming algorithm for the extended edit distance problem. In [17,18]  $O(kn)$  algorithms are given for the edit distance with only  $k$  allowed edit operations. Ukkonen [25] and Galil and Park [12] made similar improvements for cases of a given bound on the number of extended edit operations allowed. In a recent exciting paper, Cole and Hariharan [9] showed an  $O(n + (nk^4)/m)$  algorithm for the edit distance problem with only  $k$  allowed edit operations. Note that for small  $k$  their algorithm is linear, but for large  $k$  (in particular, the frequent case of an allowed given percentage of error) all algorithms for the edit distance and extended edit distance problem are still  $O(nm)$ .

Since the upper bound for the edit distance seemed very tough to break, attempts were made to consider the edit operations separately. Abrahamson [1] showed that the hamming distance problem, also known as the *string matching with mismatches* problem can be solved in time  $O(n\sqrt{m \log m})$ , i.e., within these time bounds one can find the hamming distance of the pattern at *every* text location. This is an asymptotic improvement over the  $O(nm)$  bound even in the worst case. Karloff [14] showed that if we desire only an approximation of the hamming distance, it can be done in time  $O(n \log m)$  (the time is also dependent on the closeness of the desired approximation). Amir et al. [3] showed that if the swap operation is isolated as the only edit operation allowed for computing the distance, the approximate string matching problem can be solved in time  $O(n\sqrt{m \log m})$ .

Returning to the string matching with mismatches problem, the situation is as follows. One can use the methods of Landau and Vishkin [19] or Galil and Giancarlo [11] to find all locations where the pattern matches with at most  $k$  mismatches in time  $O(nk)$ . One

can use Abrahamson's methods [1] to find the hamming distance at *every* location in time  $O(n\sqrt{m \log m})$ . The result is that if  $k < \sqrt{m \log m}$  it is worthwhile using the algorithms of [11,19], but once  $k$  grows larger than  $\sqrt{m \log m}$ , Abrahamson's algorithm is the best, and its time is independent of  $k$ .

The contributions of this paper are twofold. On the technical side, it presents the fastest known algorithms for string matching with  $k$  mismatches. We present an algorithm that is *always faster than both* the Abrahamson algorithm and the algorithms of [11,19]. We show that one can find all locations where the pattern matches with at most  $k$  mismatches in time  $O(n\sqrt{k \log k})$ . The second part of this paper improves another aspect of string matching with mismatches, and helps with the final analysis of the  $O(n\sqrt{k \log k})$  algorithm in the first part. As mentioned previously, Cole and Hariharan [9] showed an  $O(n + (nk^4)/m)$  algorithm for the edit distance problem with only  $k$  allowed edit operations. They conjecture that indeed the bound should be  $O(n + (nk^3)/m)$ . We show that for the string matching with mismatches problem (rather than the edit distance that they use) we can solve the problem in time  $O((n + (nk^3)/m) \log k)$ .

The second, and perhaps more important contribution of this paper, is in identifying and exploiting a new technique that has been implicitly used in some recent papers [4,9]—*counting*. While we do make sophisticated new uses of convolutions and use some new aspects of string periodicity, a common thread through most proofs in this paper is counting arguments. We use pigeonholing principles to space out the number of potential pattern occurrences in the text, a fact that lowers the overall complexity of the algorithms. This idea lies at the core of the algorithms of [4,9]. However, those papers used counting as a global tool to solve their specific problems. We use counting arguments even on local levels, allowing more efficient results than seemingly possible by the other algorithmic tools used.

We believe that this new method for solving string matching problems—pruning of candidates for occurrences based on counting arguments—may actually yield efficient algorithms for many more problems. The  $k$  mismatch problem we demonstrate here is only a beginning!

## 2. Problem definition and preliminaries

- (1) Let  $a, b \in \Sigma$ . Define

$$\text{neq}(a, b) \stackrel{\text{def}}{=} \begin{cases} 1, & \text{if } a \neq b; \\ 0, & \text{if } a = b. \end{cases}$$

- (2) Let  $X = x_0x_1 \dots x_{n-1}$  and  $Y = y_0y_1 \dots y_{n-1}$  be two strings over alphabet  $\Sigma$ . Then the *hamming distance* between  $X$  and  $Y$  ( $\text{ham}(X, Y)$ ) is defined as

$$\text{ham}(X, Y) \stackrel{\text{def}}{=} \sum_{i=0}^{n-1} \text{neq}(x_i, y_i).$$

- (3) The *String Matching with  $k$  Mismatches* problem is defined as follows:

**Input:** Text  $T = t_0 \dots t_{n-1}$ , pattern  $P = p_0 \dots p_{m-1}$ , where  $t_i, p_j \in \Sigma$ ,  $i = 0, \dots, n-1$ ;  $j = 0, \dots, m-1$ , and a natural number  $k$ .  
**Output:** All pairs  $\langle i, \text{ham}(P, T^{(1)}) \rangle$ , where  $i$  is a text location for which  $\text{ham}(P, T^{(i)}) \leq k$ , where  $T^{(i)} = t_i t_{i+1}, \dots, t_{i+m-1}$ .

Landau and Vishkin [19] introduced a method of using suffix trees (see, e.g., [22,28]) and Lowest Common Ancestor (see, e.g., [7,13]) in order to allow constant-time “jumps” over equal substrings in the text and pattern. The method is known as the kangaroo method. Galil and Giancarlo [11] utilized this method for string matching with  $k$  mismatches as follows. Since we are interested only in locations with at most  $k$  errors, we can simply start at each text location, and check how many mismatches there are. Every mismatch takes time  $O(1)$ , since we cover the longest equal substring and land on the next mismatch. If a location has more than  $k$  mismatches, we stop. Thus, verification of every location takes time  $O(k)$  for a total of  $O(nk)$ . Abrahamson [1] gave an algorithm that finds  $\text{ham}(P, T^{(i)}) \forall i$  in total time  $O(n\sqrt{m \log m})$ .

In the next sections we present algorithms that solve the pattern matching with  $k$  mismatches problem in time  $O(n\sqrt{k \log k})$ .

### 3. Large and small alphabets

We start with a special case that helps illustrate some of the concepts of our algorithm. This case assumes a *large* alphabet, specifically, one where the number of *different* alphabet symbols in the pattern exceeds  $2k$ . We will show that we can find all pattern occurrences in the text with no more than  $k$  mismatches, in linear time.

The algorithm has two stages. The *marking* stage and the *verification* stage. In the marking stage we identify potential starts of the pattern, and do a crude pruning of the potential candidate. A counting argument shows that we are left with at most  $n/k$  potential candidates. In the verification stage we verify which of the potential candidates is indeed a pattern occurrence. This verification is done in time  $O(k)$  per candidate, making the total time linear.

#### 3.1. The marking stage

Let  $\{a_1, \dots, a_{2k}\}$  be  $2k$  different alphabet symbols appearing in the pattern and let  $i_j$  be the smallest index in the pattern where  $a_j$  appears,  $j = 1, \dots, 2k$ . In other words, for all  $j = 1, \dots, 2k$ ,  $a_j = p_{i_j}$  and  $a_j \neq p_\ell$ ,  $\ell = 1, \dots, i_j - 1$ .

- M.1. For every text symbol  $t_i$ : If  $t_i = a_j$  then mark text location  $i - i_j$ .  
M.2. Discard every text location that is marked less than  $k$  marks.

**Time:**  $O(n)$ .

Note that the elements we are discarding cannot be starts of pattern occurrences since each of them has at least  $k$  errors (of the  $2k$  pattern elements  $\{a_1, \dots, a_{2k}\}$  at least  $k$  did not match their corresponding text position, otherwise there would have been more marks).

**Lemma 1.** *At the conclusion of the marking stage there are at most  $n/k$  undiscarded locations.*

**Proof.** Since the algorithm makes a total of  $n$  marks and since every undiscarded location has at least  $k$  marks, it means that at most  $n/k$  locations are undiscarded.  $\square$

### 3.2. The verification stage

We use the kangaroo method [11,19] described in Section 2. Verification takes time  $O(k)$  for every candidate location.

*Time:* From Lemma 1 we get that there are at most  $n/k$  candidate locations, thus the total verification time is  $O(n)$ .

We now consider the case where  $P$  has a very *small* alphabet, e.g., less than  $2\sqrt{k}$  different alphabet symbols. We will use the well-known method of convolutions, as introduced by Fischer and Paterson [10]. We need some definitions first.

Define

$$\chi_\sigma(x) = \begin{cases} 1 & \text{if } x = \sigma, \\ 0 & \text{if } x \neq \sigma, \end{cases} \quad \chi_{\bar{\sigma}}(x) = \begin{cases} 1 & \text{if } x \neq \sigma, \\ 0 & \text{if } x = \sigma. \end{cases}$$

If  $X = x_0 \dots x_{n-1}$  then  $\chi_\sigma(X) = \chi_\sigma(x_0) \dots \chi_\sigma(x_{n-1})$ . Similarly, define  $\chi_{\bar{\sigma}}(X)$ . For string  $S = s_0 \dots s_{n-1}$ ,  $S^R$  is the reversal of the string, i.e.,  $s_{n-1} \dots s_0$ .

We return to the mismatch problem for small alphabets. The product  $\chi_{\bar{\sigma}}(T)$  by  $\chi_\sigma(P^R)$  is an array where the number in each location is the number of mismatches of a non- $\sigma$  text element with a  $\sigma$  in the pattern. If we multiply  $\chi_{\bar{\sigma}}(T)$  by  $\chi_\sigma(P)^R$ , for every  $\sigma \in \Sigma$ , and add the results, we get the total number of mismatches. Since polynomial multiplication can be done in time  $O(n \log m)$  using FFT, and we do  $|\Sigma|$  multiplications, the total time for finding all mismatches using this scheme is  $O(|\Sigma|n \log m)$ .

*Time:* Our alphabet size is  $O(\sqrt{k})$ , so the problem can be solved in time  $O(n\sqrt{k} \log m)$ .

## 4. General alphabets

We are now dealing with the cases where the size of the pattern alphabet is between  $2\sqrt{k}$  and  $2k$ .

**Definition.** A symbol that appears in the pattern at least  $\sqrt{k}$  times is called *frequent*. A symbol that is not frequent is called *rare*.

We consider two cases, where there exist at least  $\sqrt{k}$  frequent symbol, and where the number of frequent symbols is smaller than  $\sqrt{k}$ . We begin with the large number of frequent symbols.

#### 4.1. Many frequent symbols

The following lemma is a counting lemma with a similar flavor to Lemma 1. We will encounter this counting argument several more times throughout this paper.

**Lemma 2.** *Let  $\{a_1, \dots, a_{\sqrt{k}}\}$  be frequent symbols. Then there exist in the text at most  $(2n)/\sqrt{k}$  locations where there is a pattern occurrence with no more than  $k$  errors.*

**Proof.** By counting. Choose  $2\sqrt{k}$  occurrences of every frequent symbol and call them the *relevant* occurrences. For every text element  $t_i$ , mark all locations where a pattern occurrence would match  $t_i$ , in case  $t_i$  is one of the frequent symbols  $\{a_1, \dots, a_{\sqrt{k}}\}$  and the match is one of the relevant occurrences. In other words, we mark all locations  $i - j$  where  $t_i = p_j$ ,  $t_i \in \{a_1, \dots, a_{\sqrt{k}}\}$ , and  $p_j$  is a relevant occurrence of  $t_i$ .

The total number of marks we made is at most  $2n\sqrt{k}$ . However, this count may include many possible overlaps. The only cases that interest us are those where no more than  $k$  errors occur. Consider a fixed text location as a start of a pattern occurrence. If more than  $k$  of our  $\sqrt{k}$  frequent symbols and their  $2\sqrt{k}$  relevant occurrences are mismatches, then there clearly does not exist a pattern occurrence with less than  $k$  mismatches. Thus, any text location with less than  $k$  marks, cannot be a pattern occurrence.

Since the total number of marks is  $2n\sqrt{k}$  and each potential pattern occurrence must have at least  $k$  marks, it leaves us with at most  $(2n\sqrt{k})/k = (2n)/\sqrt{k}$  candidates.  $\square$

*Verification:* Each of these  $O(n/\sqrt{k})$  location can be verified in time  $O(k)$  per location as described in the verification stage of Section 3 for a total  $O(n\sqrt{k})$  time.

*Finding the potential locations:* We need to show that the  $(2n)/\sqrt{k}$  potential pattern starts can be found in efficient time. We make use of the following result.

Define the *mismatch problem with “don’t cares”* as follows. Let  $T$  be a text of size  $n$  and  $P$  a pattern of size  $m$  where  $g$  pattern elements are from  $\Sigma$  and the rest are  $\phi$  (“don’t care”). Find, for every text location  $i$ , the number of mismatches between the length  $m$  text substring starting at  $i$ , and the pattern. In Section 5 we show how to solve the mismatch problem with “don’t cares” can be solved in time  $O(n\sqrt{g \log m})$ .

For our purposes, construct a new pattern  $P'$  that is equal to  $P$  in all  $2k$  location of the  $2\sqrt{k}$  relevant occurrences of each of the  $\sqrt{k}$  frequent symbols, and has “don’t care” symbols in all other locations. Using the algorithm of [3] for the mismatch problem with “don’t cares,” we can find the number of mismatches of  $P'$  in every location of  $T$  in time  $O(n\sqrt{k \log m})$ . An additional linear run will allow us to eliminate all locations with number of mismatches exceeding  $k$ , thus finding all required  $(2n)/\sqrt{k}$  candidates.

#### 4.2. Few frequent symbols

The last remaining case is where there are less than  $\sqrt{k}$  frequent symbols. We can check the number of mismatches contributed by each frequent symbol separately, by using convolutions as described in Section 3 in time  $O(n\sqrt{k} \log m)$ . Now replace all frequent symbols in  $P$  by “don’t cares.”

Two cases remain:

- (1) The remaining symbols and all their occurrences together number less than  $2k$ . Since all other elements of  $P$  are now “don’t care,” we can use the algorithm in [3] to conclude our algorithm in time  $O(n\sqrt{k} \log m)$ .
- (2) The remaining symbols and their occurrences number at least  $2k$ . Choose any  $2k$  symbols. Because all remaining symbols are not frequent, the number of occurrences of any chosen symbol does not exceed  $2\sqrt{k}$ . We now proceed as in the case of the many frequent symbols in Section 4.1. The total number of marks is at most  $n2\sqrt{k}$  (since all symbols are rare). Lemma 2 assures us that there are at most  $O(n/\sqrt{k})$  potential pattern occurrences. Finding the occurrences and verifying them is done as in Section 4.1 in total time  $O(n\sqrt{k} \log m)$ .

We have shown that we can solve the string matching with  $k$  mismatches problem in time  $O(n\sqrt{k} \log m)$ . With the aid of the next section we will show that this bound can be improved to  $O(n\sqrt{k} \log k)$ . The reason is as follows. We will see an algorithm for the string matching with  $k$  mismatches problem whose running time is  $O((n + (nk^3)/m) \log k)$ . For  $k < m^{1/3}$ , this algorithm runs in time  $O(n \log k)$ , thus it is clearly superior. For  $k > m^{1/3}$ ,  $\log m = O(\log k)$ , therefore, the algorithm we have just seen runs in time  $O(n\sqrt{k} \log k)$ .

### 5. Mismatch with don’t cares problem

The *mismatch with don’t cares problem* is defined as follows:

**Input:** Text string  $T = t_0 \dots t_{n-1}$  and pattern string  $P = p_0 \dots p_{m-1}$  where  $t_i, p_i \in \Sigma \cup \{\phi\}$ .

**Output:**  $\langle i, \text{ham}_\phi(P, T^{(i)}) \rangle$  for each location  $i$ , where for strings  $X = x_0 \dots x_{n-1}$  and  $Y = y_0 \dots y_{n-1}$

$$\text{ham}_\phi(X, Y) \stackrel{\text{def}}{=} \sum_{i=0}^{|X|-1} \text{neq}_\phi(x_i, y_i) \quad \text{and} \quad \text{neq}_\phi(a, b) \stackrel{\text{def}}{=} \begin{cases} 1, & \text{if } a \neq \phi, b \neq \phi; \\ & \text{and } a \neq b; \\ 0, & \text{otherwise.} \end{cases}$$

Assume that there are  $g$  characters in the pattern that are *not*  $\phi$  (i.e.,  $m - g$  characters are  $\phi$ ). A minor adjustment to the algorithm of [1] allows us to solve the mismatch with don’t cares problem in time  $O(n\sqrt{g} \log m)$ .

**Notation.** For  $\sigma \in \Sigma$ ,  $x \in \Sigma \cup \{\phi\}$  let

$$\chi_\sigma(x) = \begin{cases} 1 & \text{if } x = \sigma, \\ 0 & \text{if } x \neq \sigma, \end{cases} \quad \chi_{\neq\sigma}(x) = \begin{cases} 1 & \text{if } x \neq \sigma \text{ and } x \neq \phi, \\ 0 & \text{otherwise.} \end{cases}$$

If  $X = x_0 \dots x_{n-1}$   $\chi_\sigma(X) = \chi_\sigma(x_0) \dots \chi_\sigma(x_{n-1})$  and  $\chi_{\neq\sigma}(X) = \chi_{\neq\sigma}(x_0) \dots \chi_{\neq\sigma}(x_{n-1})$ .

We would like to count each non- $\phi$  character of the pattern, where it is lined up with something different than it. We can achieve this by computing, for each  $\sigma$  in  $P$ ,  $\chi_{\neq\sigma}(T) \otimes \chi_\sigma(P^R)$  (where  $\otimes$  is polynomial multiplication).

Let  $\{\sigma_1, \sigma_2, \dots, \sigma_j\}$  be the set of all different characters appearing in  $P$ . Let  $M_i = \chi_{\neq\sigma_i}(T) \otimes \chi_{\sigma_i}(P^R)$  (where  $\otimes$  is polynomial multiplication). Then  $M_i$  at position  $t$  is the number of mismatches of  $\sigma_i$  in the pattern matched with something different than  $\sigma_i$  when the pattern is lined up at  $t$ . If we let  $M$  be the sum of all the  $M_i$ 's then  $M[t]$  contains the number of mismatches when the pattern is lined up at  $t$ . By using FFT we can calculate each of the polynomial multiplications in time  $O(n \log m)$  (assuming a computer word of  $O(\log m)$  bits), for a total of  $O(jn \log m) = O(gn \log m)$ .

Abrahamson [1] uses the multiplication technique on a limited alphabet of size  $\sqrt{m}$  and then “fine tune’s” by another method. We show how to adapt this so that the limited alphabet is of size  $\sqrt{g}$  the “fine tuning” algorithm works on blocks of size  $O(\sqrt{g})$ .

Let  $P = p_0 \dots p_{m-1}$  and  $T = t_0 \dots t_{n-1}$  be the input to the algorithm. We assume that the text alphabet is the same as the pattern alphabet. This assumption can be easily achieved by replacing every text character not in the pattern alphabet by one special character, which we may add to the alphabet of the pattern.

We are now ready to present the algorithm (see Algorithm 1).

*Correctness:* The vector  $M$  is now correct since the first part of the algorithm included all the errors between blocks and the last part found all the errors within a block.

*Time:*  $O(g \log g)$  for sorting;  
 $O(n\sqrt{g} \log m)$  for counting mismatches of the representatives;  
 $O(n\sqrt{g})$  for correcting mismatches within blocks.

*Total:*  $O(n\sqrt{g} \log m)$ .

## 6. Fast algorithms for small $k$

Throughout the rest of this paper, we will assume that the length of the text is  $n = 2m$ , where  $m$  is the pattern length. This assumption is common to many solutions of string matching problems. It is justified because one may split the text into  $n/m$  substrings of length  $2m$ , with suitable overlaps so that every pattern occurrence wholly appears in some substring. One then solves the problem for each such substring of the text separately. An algorithm running in  $O(f(m, k))$  time for  $2m$  length substrings of the text immediately yields an algorithm of  $O(n/m \cdot f(m, k))$  time.



### 6.1. Overview of the algorithm for small $k$

Throughout this section we give some definitions and an outline of the method. In general, we partition the pattern into separate regions of periodic and aperiodic substrings (periodicity is defined immediately below). If we have lots of aperiodic regions we use these to accrue many mismatches. In other words, each aperiodic stretch of length  $l$  will imply that for each location where the stretch matches the next  $l/2$  locations need to have at least one mismatch. A sufficient number ( $2k$ ) of aperiodic regions of sufficient length implies that there are not too many matches near each match. On the other hand, if there are not too many aperiodic stretches, the periodic stretches become of more interest. Here we can utilize the periodicity properties.

In this section we give the necessary definitions of periodicity, “breaks” (aperiodic stretches), etc., and we show properties related to these definitions. In Section 7, we show the tradeoffs described above, in a somewhat more entailed format.

1. For every  $a \in \Sigma$  that appears more than  $\sqrt{g}$  times in  $P$  (a *frequent character*), use FFT to count all text locations where there is a text character that is different from its corresponding pattern character.
2. Let  $p_{j_0}, \dots, p_{j_{g_1}}$  be the non-frequent, non- $\phi$  characters of  $P$ . Consider  $L = \langle p_{j_0}, j_0 \rangle, \langle p_{j_1}, j_1 \rangle, \dots, \langle p_{j_{g_1}}, j_{g_1} \rangle$ . (Every non- $\phi$  pattern character has a corresponding pair  $\langle s, d \rangle$  where  $s$  is the character and  $d$  is its location in  $P$ .)
3. Sort  $L$  lexicographically. (There are no more than  $g$  pairs in  $L$ .) Call the sorted array  $L'$ .
4. Divide  $L'$  into  $g_2 \leq \sqrt{g}$  blocks, each containing no more than  $2\sqrt{g}$  pairs, in a manner that no character appears in more than one block. (We are assured that such a division is possible because all remaining characters are non-frequent.)
5. For each block  $B_i$ ,  $i = 0, \dots, g_2 \leq \sqrt{g}$ , let  $b_i = \langle s, d \rangle$  be the smallest (leftmost) pair in the block; call  $s$  the *representative* of the characters in the pairs in block  $B_i$ .
6. Let  $T'$  and  $P'$  be  $T$  and  $P$  such that every  $t_i$  and  $p_i$  is replaced by its representative. (Implemented by a sequential scan of  $L'$ .)
7. Count all mismatches of  $P'$  in  $T'$  (using FFT).  
( $P'$  and  $T'$  can be considered “flattened out” versions of  $P$  and  $T$ . When we seek all mismatches of  $P'$  in  $T'$  we only detect the “large” mismatches, i.e., those between characters that are so different that they are in different blocks. However, mismatches between characters of the same block are undetected. At this stage we must “fine tune” our approximate solution. We scan  $T$  and for every character  $t_i$  of  $T$  we only compare it to the  $O(\sqrt{g})$  characters of  $P$  that are in  $t_i$ ’s block.)
8. For  $i = 0$  to  $n - 1$  if  $t_i \neq \phi$  do
  - Let  $B_{t_i}$  be the block of  $L'$  that contains  $t_i$  (there must be such a block, since the pattern alphabet and the text alphabet are the same)
  - For every pair  $\langle s, d \rangle \in B_{t_i}$   
(at most  $2\sqrt{g}$  characters)
    - if  $t_i \neq s$  then  $M[i - d] \leftarrow M[i - d] + 1$
  - end

Algorithm 1.

## 6.2. Periodicity, aperiodicity, and breaks

**Periodicity.** A string  $S[1 \dots n]$  is *periodic* if  $\exists i \leq n/2$  such that  $\forall j \leq n - i + 1$   $S[j] = S[i + j - 1]$ . Alternatively,  $S$  is periodic if  $S = u^j w$ , where  $j \geq 2$  and  $w$  is a prefix of  $u$ . We say that the period of  $S$  is  $u$ . A string is *aperiodic* if it is not periodic.

Periodicity of strings is central to many of the algorithms for exact string matching, e.g., [15,19,26]. In a recent paper by Cole and Hariharan [9] periodicity was utilized for approximate string matching, where the distance measure is the classical edit distance.

Note that if the pattern is periodic with a short period it is quite simple to come up with a quick algorithm for string matching with  $k$  mismatches. Even if there is a short substring of the pattern whose removal would split the pattern into two substrings, each of which is periodic, a fast algorithm could be devised. In [9] these substrings, whose removal splits the pattern into periodic substrings, were found to be useful for approximate string matching. This brings us to the following definition.

**Definition.** A *break* of a string  $S$  is an aperiodic substring of  $S$ . An  *$l$ -break* is a break of length  $l$ .

Few  $l$ -breaks, where  $l$  is small, would seem ideal for fast algorithms. It would seem that the more breaks there are the more complicated things get. However, it turns out that a large number of breaks also creates a lot of mismatches. In fact, a large number of breaks can be very useful in the design of fast algorithms for string matching with  $k$  mismatches.

**Lemma 3.** Let  $P$  be a pattern with  $2k$  disjoint  $l$ -breaks and let  $T$  be a text. In each match of  $P$  in  $T$  at least  $k$  of the  $l$ -breaks match exactly.

**Proof.** In a match of  $P$  in  $T$  there are at most  $k$  mismatches. Yet, there are  $2k$  disjoint  $l$ -breaks. Since at most  $k$  do not match exactly, at least  $k$  must match exactly.  $\square$

Consider a pattern in which we have identified as many  $l$ -breaks as possible. Between each of these  $l$ -breaks there is a stretch which must be periodic with a period of size  $\leq l/2$ . We call these stretches *periodic stretches*. Figure 1 illustrates the form of a pattern with its breaks and periodic stretches.

Cole and Hariharan [9] give a method for easily finding the breaks in linear time.

The following lemma uses the fact mentioned earlier that the text  $T$  is of length  $2m$ . Its proof closely follows Section 6 from [9].

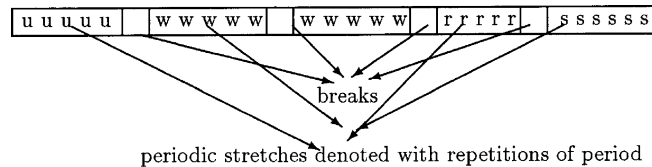


Fig. 1.

**Lemma 4.** *Let  $P$  be an  $m$  length pattern with less than  $2k$   $l$ -breaks. Let  $T$  be of length  $2m$ . Then all matches of  $P$  in  $T$  are in a substring of  $T$  which has at most  $O(k)$   $l$ -breaks.*

### 6.3. Counting arguments

As in Sections 3 and 4, the main power of this algorithm also stems from pigeonhole reasons. We present below the necessary counting arguments.

**Theorem 1.** *Let  $P$  be a pattern with  $2k$  disjoint  $l$ -breaks and let  $T$  be a text. In every  $l$  contiguous locations in  $T$  there are at most 4 matches of the pattern.*

**Proof.** By Lemma 3 for the pattern to match at a given location at least  $k$  of the  $l$ -breaks must match exactly. However, an  $l$ -break  $B$  is aperiodic. So, the distance between two exact matches of  $B$  in the text is at least  $l/2$ . It follows that for  $l$  contiguous locations in  $T$ , the overall number of exact matches of the  $l$ -breaks of  $P$ , in their respective locations, is at most  $4k$ . This means that at most 4 locations have  $k$   $l$ -breaks with an exact match, in their respective locations.  $\square$

**Corollary 1.** *If  $P$  has  $2k$  disjoint  $k$ -breaks then there are at most  $(4n)/k$  matches of  $P$  in  $T$ . These matches can be found in  $O(n + m)$  time.*

**Proof.** It directly follows from Theorem 1 that there are at most  $(4n)/k$  matches of  $P$  in  $T$ . Therefore, if we knew these locations in advance, verification would take  $O(k)$  time per location, as described in the verification stage of Section 3.  $\square$

We now describe a method of finding the  $O(n/k)$  candidate locations in time  $O(n)$ . This method is similar to the marking stage of Section 3. We first set up some terminology that will be useful in future discussions.

**Terminology.** Let  $X$  be a substring of the pattern that starts at location  $j$  of the pattern. Assume  $X$  appears in location  $i$  of the text. We call location  $i - j$  of the text the *text location for the pattern occurrence appropriate for  $X$* , i.e.,  $i - j$  is a text location where, if the pattern were to start, it will reach substring  $X$  in location  $i$  of the text.

We will be concerned with the pattern occurrences appropriate for breaks. Note that if several different breaks are the same substring, then an appearance of that substring in a text location means several different locations for appropriate pattern occurrences.

We are now ready to describe our marking stage.

1. Find all exact matches of all breaks in the text.
2. For every such match, mark all text locations for pattern occurrences appropriate for this break.
3. Discard every text location that is marked less than  $k$  marks.

This is, in essence, the marking stage of Section 3. We need to show the following:

- (1) There are  $O(n)$  exact matches of breaks and they can be found in linear time.
- (2) There is a total of  $O(n)$  marks.

Both above claims are true for the following reasons.

- (1) Each  $k$ -break is aperiodic so there is at least a  $k/2$  distance between two different appearances of a given break in the text. Since each distinct  $k$ -break can appear at most  $(2n)/k$  times in the text, and since there are  $2k$   $k$ -breaks, the overall number of possible matches of all  $k$ -breaks in the text does not exceed  $4n$ . In addition, the total length of all  $k$ -breaks does not exceed  $m$ , since they are all disjoint substrings of the pattern. All exact matches of all  $k$ -breaks in the text can therefore be found in time  $O(n + m)$  by a number of methods (e.g., [2,11]).
- (2) The total number of marks may not exceed  $4n$  for the following reason. Suppose there are  $\ell$  distinct breaks, appearing  $a_1, \dots, a_\ell$  times, respectively. Since the total number of appearances of each distinct  $k$ -break does not exceed  $(2n)/k$ , the total number of marks does not exceed  $(2n)/k \sum_{i=1}^{\ell} a_i \leq 4n$ .

It follows from the corollary that when there are  $2k$   $k$ -breaks in the pattern we can solve the problem quickly. However, often the pattern does not contain  $2k$   $k$ -breaks. Nevertheless, there may be an  $l$  such that there are  $2k$   $l$ -breaks. We know from Theorem 1 that there are at most  $n/l$  matches of the pattern in the text. The ideas we will present later are based on searching for these matches. Yet finding them may be too costly since using the method for sparsifying the match locations described in the proof of the previous corollary will take  $O((nk)/l)$  time. To circumvent this problem, rather than searching for all matches, we need a way to seek for local matches. The following lemma gives precisely such a method.

**Lemma 5.** *Let  $P$  be a pattern with  $2k$  disjoint  $l$ -breaks and let  $T$  be a text of size  $n$ . We can preprocess  $T$  in  $O(n)$  time such that, given  $l$  contiguous text locations, we can identify the, at most 4, locations where  $P$  matches in time  $O(k \log k)$ .*

**Proof.** Let  $S = \{B_1, \dots, B_{2k}\}$  be a set of  $2k$  disjoint  $l$ -breaks of  $P$ . Let  $S' = \{B'_1, \dots, B'_f\}$ , where  $f \leq 2k$ , be the maximal subset of distinct  $l$ -breaks of  $S$ , i.e.,  $\forall i, j \ B'_i \neq B'_j$  and  $\forall i \exists j \ B_i = B'_j$ .  $S'$  can be found in  $O(\sum_{i=1}^{2k} |B_i|) = O(m)$  time by constructing a tree of the strings in  $S$ .

Note that since each break in  $S'$  is distinct, the overall number of exact matches of  $l$ -breaks of  $S'$  in  $T$  is bounded by  $n$ , the length of  $T$ . These exact matches can be found in  $O(n + \sum_{i=1}^f |B'_i|) = O(n + m)$  time [2,11].

Consider an array  $A$  of length  $n$ , corresponding to the  $n$  locations of the text, with  $A[i]$  containing the index of the  $l$ -break of  $S'$  that exactly matches at location  $i$  of  $T$ , if any. Partition this array, into  $n/k$  pieces of size  $k$ , i.e.,  $A[1] \dots A[k]$ ,  $A[k+1] \dots A[2k]$ ,  $\dots$ . To simplify matters later on, we partition the array once more into another  $n/k$  pieces of size  $k$  that overlap the previous, i.e.,  $A[k/2+1] \dots A[(3k)/2]$ ,  $A[(3k)/2+1] \dots A[(5k)/2]$ ,  $\dots$ .

For each piece of size  $k$  and each break  $B'_j$  in  $S'$  create a balanced binary search tree with leaves corresponding to the locations containing  $j$  in this piece of size  $k$ .

The number of trees created is  $(2n)/k \cdot f \geq (2n)/k \cdot 2k = 4n$ . The size of each tree is  $O(1) + O(\text{number of leaves})$ . The leaves of all the trees together correspond to all the exact matches of the  $l$ -breaks of  $S'$  in  $T$ . Since there are at most  $n$  such exact matches, it follows that the overall size of the trees is  $O(n)$ . It is straightforward to see that the trees can be constructed in  $O(n)$  time.

In order to identify the, at most 4, locations where  $P$  matches in  $l$  contiguous text locations we utilize these trees. Lemma 3 tells us that a match of the pattern dictates that  $k$  out of the pattern's  $l$ -breaks  $B_1, \dots, B_{2k}$  match exactly in  $T$  at their appropriate shift. However, by definition of  $l$ -breaks each  $B_i$  is aperiodic and therefore there is distance of at least  $l/2$  between exact matches of  $B_i$  in  $T$ . Hence, each  $B_i$  contributes at most 2 exact matches to any of the potential matches of the pattern in the  $l$  contiguous text locations. Since the 2 exact matches of  $B_i$  must also appear in  $l \leq k$  contiguous text locations, we can find them by using exactly one of the binary search trees described above. The tree is balanced therefore finding the 2 exact matches takes  $O(\log k)$  time. Finding the exact matches of all  $B_i$  and marking the potential matches accordingly takes overall  $O(k \log k)$  time with at most an overall  $2 \cdot 2k = 4k$  marks. Since only 4, out of the  $l$ , locations for potential matches can have  $k$  marks, the pattern can match at most 4 locations. These 4 potential locations can be verified for a match in  $O(k)$  time [6,11].  $\square$

#### 6.4. The $l$ -boundary of a pattern

Lemma 5 gives us a good handle on dealing with many breaks. But this is not sufficient on its own. In fact, as we have previously mentioned, few breaks is also a promising start for devising fast algorithms. The core of the idea in the sections ahead is to merge the advantages of many breaks and few breaks.

**$l$ -boundary.** Clearly, the number of  $l$ -breaks in a pattern  $P$  is at most the number of  $(l-1)$ -breaks in  $P$ . We define the  $l$ -boundary of a pattern  $P$  to be the  $l$  such that  $P$  contains fewer than  $2k$   $l$ -breaks and at least  $2k$   $(l-1)$ -breaks. If there are at least  $2k$   $k$ -breaks or if there are fewer than  $2k$  2-breaks then the  $l$ -boundary is undefined.

The  $l$ -boundary of a pattern is exactly the point where we get the best of both worlds. We have seen the usefulness of many  $(l-1)$ -breaks. Now we make a claim regarding few  $l$ -breaks.

By the definition of  $l$ -boundary, there are at least  $2k$   $(l-1)$ -breaks. Therefore, we know, by Theorem 1, that there are at most  $O(n/l)$  matches of the pattern in the text. Moreover, we have a way to locally check this using Lemma 5. On the other hand, there are fewer than  $2k$   $l$ -breaks. Combining this with Lemma 4 the situation is that there are at most  $O(k)$   $l$ -breaks in the pattern and  $O(k)$   $l$ -breaks in the (relevant) text.

Note that the  $l$ -boundary may be found by performing a binary search on the potential  $l$ 's, where  $1 < l < k$ . As we mentioned earlier,  $l$ -breaks can be found in  $O(m)$  time [9]. Therefore, the search takes  $O(m \log k)$  time. We can improve this to  $O(m)$  time by utilizing the special structure of the pattern. The idea is to consider the original split into  $k$ -breaks

and periodic stretches and note that each  $w$ -periodic stretch containing  $i$  periods of  $w$  contributes  $i/2 (2|w| - 1)$ -breaks which are not  $2|w|$ -breaks. This follows from the fact that  $w$  is the period of this stretch, hence any  $(2|w| - 1)$  substring is aperiodic. Since there were  $i$  periods of  $w$  there are now  $i/2$  breaks of length  $(2|w| - 1)$ .

## 7. Finally—the algorithm

All matches of a pattern containing at least  $2k$  disjoint  $k$ -breaks can be found in linear time by Corollary 1. So the case to be considered is when the pattern contains fewer than  $2k$   $k$ -breaks. In this case we use the  $l$ -boundary. If there is no  $l$ -boundary, i.e., there are not even  $2k$  2-breaks then the problem can be solved in a straightforward technical manner in  $O(k^3)$  time.

Assume that we have found the  $l$ -boundary. Let  $w$  be a string of size  $|w| \leq l/2$ . Let  $w^*$  denote the infinite string  $www \dots$  and let  $w_{2l}^*$  denote its  $2l$  length prefix.

**Definition.** A string of size  $l$ ,  $s_1 \dots s_l$ , has *general period*  $w$  if it is a substring of  $w_{2l}^*$ .

Consider a partition of the pattern  $P$  into equal segments of size  $l$ , and call each such segment an  $l$ -segment. We say that  $w$  *dominates*  $P$ , or that  $w$  is a *dominating period* of  $P$ , if at most  $4k$  segments do not have general period  $w$ . A pattern that has a dominating period is said to be a *dominated pattern*.

### 7.1. Dominated patterns

Consider a string  $S$  which has at most  $2k$   $l$ -breaks. Let  $S$  be partitioned into  $l$ -segments. We say that an  $l$ -segment, which is not fully within a periodic stretch of  $S$  which has period  $w$ , is a *bad  $l$ -segment*. Note that an  $l$ -segment may have a general period  $w$  and still be a bad  $l$ -segment. For example, if a break intersects an  $l$ -segment. The following lemma shows that there cannot be too many such cases.

Remember that the pattern has an  $l$ -boundary. So, the pattern is now (1) with fewer than  $2k$   $l$ -breaks and (2) has a dominating period  $w$ . This yields the following.

**Lemma 6.** *Let  $P$  be a pattern with a dominating period  $w$ . In the partition of  $P$  into  $l$ -segments there are at most  $8k$  bad  $l$ -segments.*

**Proof.** If the  $l$ -segment is within a  $w$ -periodic stretch then by definition it is not a bad  $l$ -segment.

On the other hand, if the  $l$ -segment is within a periodic stretch that does not have period  $w$  then by definition the segment cannot have period  $w$ . The only other  $l$ -segments that may have period  $w$  but are not within  $w$ -periodic stretches are those that intersect an  $l$ -break. But, since there are fewer than  $2k$   $l$ -breaks and there are at most 2  $l$ -segments that overlap any given  $l$ -break, there are at most  $4k$   $l$ -segments that have period  $w$  but are not within  $w$ -periodic stretches. Since  $w$  is a dominating period, there are at most  $4k$   $l$ -segments that do not have period  $w$  and together with these at most  $4k$  breaks that do have period  $w$  but

```

1. Find all matches of  $P$  in  $T$  at overlapping locations.
2. For each bad  $l$ -segment  $B$  do pattern matching with mismatches, with pattern  $B$  and text  $w_{2l}^*$ .
3. Do pattern matching with mismatches, with pattern  $w$  and text  $w_{2l}^*$ .
4. Compute the # of mismatches of  $P$  at the first  $|w|$  locations of  $T$  using steps 2 and 3.
5.  $i \leftarrow |w| + 1$ .
6. While end of text not reached
    6a. if  $i$  is not an overlapping location
        6aa. # of mismatches at location  $i \leftarrow$  # of mismatches at location  $i - |w|$ ,
        6ab.  $i \leftarrow i + 1$ ;
    6b. else, if  $j$  is the next non-overlapping location
        6ba. for each of the bad  $l$ -segment that participate in an overlap in the overlapping locations (bad
            segment vs. bad segment) from  $i$  to  $j$ ,
            update the # of mismatches it accrues in the next  $|w|$  locations,
        6bb.  $i \leftarrow j$ .

```

Algorithm 2.

are not within  $w$ -periodic stretches. Therefore, there are at most  $8k$   $l$ -segments not within  $w$ -periodic stretches.  $\square$

Following Lemma 4 there are at most  $O(k)$   $l$ -breaks in the text as well. Using the proof of the previous lemma it is easy to show the following.

**Lemma 7.** *Let  $P$  be a pattern with a dominating period  $w$ . All matches of  $P$  in the text  $T$  appear in a substring of the text,  $S$ , where  $S$  has at most  $O(k)$  bad  $l$ -segments in its partition into  $l$ -segments of size  $l$ .*

It follows from the previous two lemmas that the pattern and the text both have at most  $O(k)$  bad  $l$ -segments in their partitions. This suggests the following idea. We say that a location  $i$  in the text is *overlapping* if, when placing  $P$  above  $T^{(i)}$ , some bad  $l$ -segment of the pattern overlaps some bad  $l$ -segment of the text. Since there are at most  $O(k)$  bad  $l$ -segments in both the text and the pattern it follows that there are at most  $O(k^2 \cdot l)$  overlapping locations. In fact, there are  $O(k^2)$  sets of  $2l$  contiguous overlapping locations. Applying Lemma 5 to each of these sets yields an algorithm for finding all matches of  $P$  in  $T$  at overlapping locations.

Once we have found the matches at overlapping locations, we know that at all other locations  $i$  placing  $P$  above  $T^{(i)}$  results in the bad  $l$ -segments of the pattern to be over  $w$ -periodic stretches of the text and the bad  $l$ -segments of the text to be under  $w$ -periodic stretches of the pattern. The number of mismatches accrued by the bad  $l$ -segment is the same as the number of mismatches it accrues in  $|w|$  locations forward, assuming that the  $w$ -periodic stretch beneath it, or above it, does not end less than  $w$  characters forward. This is true for every bad  $l$ -segment. So, it follows that the number of mismatches at location  $i + |w|$  is the same as the number of mismatches at the location  $i$  unless there is a overlapping location between  $i$  and  $i + |w|$ . This suggest the algorithm (see Algorithm 2).

We assume, for simplicity of presentation, that the first  $|w|$  locations are not overlapping locations and that after a set of contiguous overlapping locations there are  $|w|$  locations that

are not overlapping locations. However, even if this is not the case the situation is almost identical.

*Correctness:* Follows immediately from the discussion above.

*Time:* Step 1 takes  $O(k^3 \log k)$  since we apply Lemma 5 to  $O(k^2)$  sets of  $O(l)$  contiguous overlapping locations. Step 2 can be done in  $O(kl^2)$  per bad  $l$ -segment for an overall  $O(k \cdot l^2)$ . Since  $l \leq k$ , step 2 can be done in  $O(k^3)$ . Step 3 can similarly be done in  $O(k^2)$  time. In step 4, we must compute the number of mismatches at each of the first  $|w|$  locations. We use step 2 to compute bad  $l$ -segments vs.  $w$ -periodic strings and step 3 to compute  $w$ -periodic strings vs.  $w$ -periodic strings. Since we have precomputed the necessary, for each of the locations we do this in constant time per bad  $l$ -segment or per periodic stretch. So this takes time  $O(k)$  per location and overall  $O(k \cdot l)$  time. Step 6ba can be done using step 2. For each bad  $l$ -segment that participated in an overlap there is  $O(l)$  work. However, since there are only  $O(k^2)$  sets of  $2l$  contiguous overlapping locations, the overall time complexity of this step in the algorithm is  $O(k^2 \cdot l)$ . All other parts run in  $O(n) = O(m)$  time.

This yields the following.

**Theorem 2.** *Let  $P$  be a dominated pattern and  $T$  a text. We can find all matches of  $P$  in  $T$  in  $O(n + m \log k + (nk^3 \log k)/m)$  time.*

### 7.2. Non-dominated patterns

Let  $P$  be a pattern of length  $m$ . Since  $P$  has at most  $2k$  breaks there exists a periodic stretch of length  $\Omega(m/k)$ . Let  $w$  be the period in this periodic stretch. Since our pattern is non-dominated, it must be the case that there is a substring  $S$  containing this period stretch and a rim to the left, or to the right, such that  $S$  will contain exactly  $2k$   $l$ -segments that do not have general period  $w$ . We call this special substring of the pattern a *sparsifying substring* of  $P$ .

Note that a sparsifying substring  $S$  is also a dominated pattern. It also has the special structure of ending (or starting) in a periodic stretch of length  $\Omega(m/k)$ . The sparsifying property of this special structure is captured in the following lemma.

**Lemma 8** (Sparsifying lemma). *Let  $P$  be a pattern dominated by  $w$  and ending in a periodic stretch of length  $s$ . Let  $T$  be a text. Then at any  $s$  contiguous locations of  $T$  there are at most  $k$  locations where  $P$  matches with at most  $k$  mismatches.*

The proof is similar to the proof of Theorem 1 and hence is omitted.

**Corollary 2.** *String matching with  $k$  mismatches is solvable in  $O(n + (nk^4 \log k)/m)$  time.*

**Proof.** Dominated patterns and patterns with  $2k$   $k$ -breaks can be solved in  $O(n + m \log k + (nk^3 \log k)/m)$  and  $O(n)$  time, respectively.



For non-dominated patterns, there exists a sparsifying substring  $S$  of length  $\Omega(m/k)$ . Since a sparsifying substring is a dominated pattern, we can find all matches of  $S$  in  $T$  with at most  $k$  mismatches in  $O(n + m \log k + (nk^3 \log k)/(m/k)) = O(n + m \log k + (nk^4 \log k)/m)$  time. Since  $S$  ends in a periodic stretch of length  $\Omega(m/k)$ , by the sparsifying lemma there are at most  $k$  matches in every  $\Omega(m/k)$  contiguous locations. Therefore, for the  $2m$  length text there are at most  $O(k^2)$  locations that match  $S$  and therefore  $O(k^2)$  locations where  $P$  matches. At each location where  $S$  matches it can be verified in  $O(k)$  time whether  $P$  matches at this location for an overall  $O(k^3)$  for a  $2m$  length text, and overall  $O((nk^3)/m)$  time for a full  $n$  length string.  $\square$

We will now show that it is not really necessary to find the  $O(k^2)$  locations where  $S$  matches. Rather, we will find  $O(k^2)$  locations that are candidates for an  $S$  match, and verify those locations in time  $O(k)$ . We will find these locations in time  $O(n \log k)$ . Note that for  $k \leq m^{1/4}$  we actually have an algorithm that runs in time  $O(n \log k)$ , so we are only interested in  $k > m^{1/4}$ , in which case  $\log m = O(\log k)$ .

Assume there is a match of  $S$  with  $k$  mismatches at location  $i$  of text  $T$ . Then the following two conditions hold:

- (1) There exist in  $S$  no more than  $k$   $l$ -segments that have general period  $w$  that match text  $l$ -segments that do not have general period  $w$ .
- (2) There exist in the text no more than  $k$   $l$ -segments that have general period  $w$  that match  $l$ -segments in  $S$  that do not have general period  $w$ .

**Lemma 9.** *Within any  $\Omega(m/k)$  contiguous text locations, there are at most  $6k$  locations of length  $l$  where both conditions above hold.*

**Proof.** Call an  $l$ -segment that does not have general period  $w$  a *black*  $l$ -segment. Assume  $i$  is a text location where the conditions hold.

This means that there are at most  $3k$  black text  $l$ -segments. We know that there are exactly  $2k$  black  $l$ -segments of  $S$ . The total number of possible pairings of black text and  $S$   $l$ -segments is  $6k^2$ . This number is achieved while all black  $l$ -segments from  $S$  are still within the area of all black text  $l$ -segments, i.e., within  $\Omega(m/k)$  contiguous text locations.

Any match of  $S$  with the text, must have at least  $k$  black  $l$ -segment pairs. Since the total number of pairs is  $6k^2$  it means that there are no more than  $6k$  possible pairings of black  $l$ -segments that satisfy both conditions. However, since the  $l$ -segments are of length  $l$  yet  $S$  may start at every text location, this means that we may identify  $6k$  starts of contiguous text locations of length  $l$  within every  $\Omega(m/k)$  contiguous text locations.  $\square$

#### 7.2.1. Finding the locations which satisfy the conditions

It is clear that any location where the above condition does not hold, cannot have an occurrence of  $S$ . Thus, if we find all locations where these conditions hold, we are done. Each condition can be ascertained using one convolution, so all locations where both conditions hold can be found in time  $O(n \log m) = O(n \log k)$ .

The convolutions may identify more than  $k$  locations per  $m/k$  block, but according to the lemma, we know that the identified locations are within  $k$  length- $l$  contiguous text

locations. By the proof of Lemma 5 we may find the at most 4 locations where the pattern may match in time  $O(k \log k)$  per candidate. Conclude: we have  $k$  segments of length  $m/k$ . In each one we find the  $4 \cdot 6 \cdot k$  potential candidates for an  $s$  occurrence in time  $O(k \log k)$  making the total time  $O(k^3 \log k)$ .

**Theorem 3.** *There is an algorithm that solves the string matching with  $k$  mismatches problem in time  $O(n \log k + (nk^3 \log k)/m)$ .*

## References

- [1] K. Abrahamson, Generalized string matching, *SIAM J. Computing* 16 (6) (1987) 1039–1051.
- [2] A.V. Aho, M.J. Corasick, Efficient string matching, *Commun. ACM* 18 (6) (1975) 333–340.
- [3] A. Amir, Y. Aumann, G. Landau, M. Lewenstein, N. Lewenstein, Pattern matching with swaps, *J. Algorithms* 37 (2000) 247–266.
- [4] A. Amir, A. Butman, M. Lewenstein, Real scaled matching, *Inform. Process. Lett.* 68 (3) (1999) 125–132.
- [5] A. Amir, M. Farach, Efficient 2-dimensional approximate matching of half-rectangular figures, *Inform. and Comput.* 118 (1) (April 1995) 1–11.
- [6] B.S. Baker, A theory of parameterized pattern matching: algorithms and applications, in: *Proc. 25th Annual ACM Symposium on the Theory of Computation*, 1993, pp. 71–80.
- [7] O. Berkman, D. Breslauer, Z. Galil, B. Schieber, U. Vishkin, Highly parallelizable problems, in: *Proc. 21st ACM Symposium on Theory of Computation*, 1989, pp. 309–319.
- [8] R.S. Boyer, J.S. Moore, A fast string searching algorithm, *Commun. ACM* 20 (1977) 762–772.
- [9] R. Cole, R. Hariharan, Approximate string matching: A faster simpler algorithm, in: *Proc. 9th ACM–SIAM Symposium on Discrete Algorithms (SODA)*, 1998, pp. 463–472.
- [10] M.J. Fischer, M.S. Paterson, String matching and other products, in: R.M. Karp (Ed.), *Complexity of Computation*, in: *SIAM–AMS Proceedings*, vol. 7, 1974, pp. 113–125.
- [11] Z. Galil, R. Giancarlo, Improved string matching with  $k$  mismatches, *SIGACT News* 17 (4) (1986) 52–54.
- [12] Z. Galil, K. Park, An improved algorithm for approximate string matching, *SIAM J. Comput.* 19 (6) (1990) 989–999.
- [13] D. Harel, R.E. Tarjan, Fast algorithms for finding nearest common ancestor, *Comput. Syst. Science* 13 (1984) 338–355.
- [14] H. Karloff, Fast algorithms for approximately counting mismatches, *Inform. Process. Lett.* 48 (2) (1993) 53–60.
- [15] D.E. Knuth, J.H. Morris, V.R. Pratt, Fast pattern matching in strings, *SIAM J. Comput.* 6 (1977) 323–350.
- [16] G.M. Landau, U. Vishkin, Efficient string matching with  $k$  mismatches, *Theoret. Comput. Sci.* 43 (1986) 239–249.
- [17] G.M. Landau, U. Vishkin, Fast parallel and serial approximate string matching, *J. Algorithms* 10 (2) (1989) 157–169.
- [18] G.M. Landau, E.W. Myers, J.P. Schmidt, Incremental string comparison, *SIAM J. Comput.* 27 (2) (1998) 557–582.
- [19] G.M. Landau, U. Vishkin, Efficient string matching in the presence of errors, in: *Proc. 26th IEEE FOCS*, 1985, pp. 126–136.
- [20] V.I. Levenshtein, Binary codes capable of correcting, deletions, insertions and reversals, *Soviet Phys. Dokl.* 10 (1996) 707–710.
- [21] R. Lowrance, R.A. Wagner, An extension of the string-to-string correction problem, *J. ACM* 22 (1975) 177–183.
- [22] E.M. McCreight, A space-economical suffix tree construction algorithm, *J. ACM* 23 (1976) 262–272.
- [23] M.V. Olson, A time to sequence, *Science* 270 (1995) 394–396.
- [24] A. Pentland, Invited talk, NSF Institutional Infrastructure Workshop, 1992.
- [25] E. Ukkonen, Algorithms for approximate string matching, *Inform. and Control* 64 (1985) 100–118.

- [26] U. Vishkin, Deterministic sampling—a new technique for fast pattern matching, *SIAM J. Comput.* 20 (1991) 303–314.
- [27] R.A. Wagner, On the complexity of the extended string-to-string correction problem, in: *Proc. 7th Annual ACM Symposium on the Theory of Computation*, 1975, pp. 218–223.
- [28] P. Weiner, Linear pattern matching algorithm, in: *Proc. 14th IEEE Symposium on Switching and Automata Theory*, 1973, pp. 1–11.