Algorithms **2015**, *2*, 248-270; doi:10.3390/a8020248

OPEN ACCESS

algorithms

ISSN 1999-4893

www.mdpi.com/journal/algorithms

Article

On String Matching with Mismatches

Marius Nicolae * and Sanguthevar Rajasekaran

Department of Computer Science and Engineering, University of Connecticut,
371 Fairfield Way Unit 4155, Storrs, CT 06269, USA; E-Mail: rajasek@engr.uconn.edu

* Author to whom correspondence should be addressed; E-Mail: marius.nicolae@uconn.edu;
Tel.: +1-860-450-6023

Academic Editor: Giuseppe Lancia

Received: 3 April 2015 / Accepted: 19 May 2015 / Published: 26 May 2015

Abstract: In this paper, we consider several variants of the pattern matching with mismatches problem. In particular, given a text $T = t_1t_2 \cdots t_n$ and a pattern $P = p_1p_2 \cdots p_m$, we investigate the following problems: (1) pattern matching with mismatches: for every $i, 1 \leq i \leq n - m + 1$ output, the distance between P and $t_it_{i+1} \cdots t_{i+m-1}$; and (2) pattern matching with k mismatches: output those positions i where the distance between P and $t_it_{i+1} \cdots t_{i+m-1}$ is less than a given threshold k. The distance metric used is the Hamming distance. We present some novel algorithms and techniques for solving these problems. We offer deterministic, randomized and approximation algorithms. We consider variants of these problems where there could be wild cards in either the text or the pattern or both. We also present an experimental evaluation of these algorithms. The source code is available at http://www.engr.uconn.edu/∼man09004/kmis.zip.

Keywords: pattern matching with mismatches; kmismatches problem; approximate counting of mismatches

---

1. Introduction

The problem of string matching has been studied extensively due to its wide range of applications from Internet searches to computational biology. String matching can be defined as follows. Given a text $T = t_1t_2 \cdots t_n$ and a pattern $P = p_1p_2 \cdots p_m$, with letters from an alphabet Σ, find all of the occurrences of the pattern in the text. This problem can be solved in $O(n + m)$ time by using well-known algorithms

(e.g., KMP[1]). A variation of this problem is to search for multiple patterns at the same time. An algorithm for this version is given in [2].

A more general formulation allows "don't care" or "wild card" characters in the text and the pattern. A wild card matches any character. An algorithm for pattern matching with wild cards is given in [3] and has a runtime of $O(n \log |\Sigma| \log m)$. The algorithm maps each character in Σ to a binary code of length $\log |\Sigma|$. Then, a constant number of convolution operations is used to check for mismatches between the pattern and any position in the text. For the same problem, a randomized algorithm that runs in $O(n \log n)$ time with high probability is given in [4]. A slightly faster randomized $O(n \log m)$ algorithm is given in [5]. A simple deterministic $O(n \log m)$ time algorithm based on convolutions is given in [6].

A more challenging formulation of the problem is pattern matching with mismatches. This formulation appears in two versions: (1) for every alignment of the pattern in the text, find the distance between the pattern and the alignment; or (2) identify only those alignments where the distance between the pattern and the text is less than a given threshold. The distance metric can be the Hamming distance, edit distance, L_1 metric, and so on. The problem has been generalized to use trees instead of sequences or to use sets of characters instead of single characters (see [7]).

A survey of string matching with mismatches is given in [8]. A description of practical on-line string searching algorithms can be found in [9].

The Hamming distance between two strings A and B, of equal length, is defined as the number of positions where the two strings differ and is denoted by $Hd(A, B)$.

In this paper, we are interested in the following two problems, with and without wild cards.

1. Pattern matching with mismatches: Given a text $T = t_1 t_2 \dots t_n$ and a pattern $P = p_1 p_2 \dots p_m$, output $Hd(P, t_i t_{i+1} \dots t_{i+m-1})$, for every $i,\ 1 \leq i \leq n - m + 1$.

2. Pattern matching with k mismatches (or the k mismatches problem): Take the same input as above, plus an integer k. Output all i, $1 \leq i \leq n - m + 1$, for which $Hd(P, t_i t_{i+1}, \dots t_{i+m-1}) \leq k$.

1.1. Pattern Matching with Mismatches

For pattern matching with mismatches, a naive algorithm computes the Hamming distance for every alignment of the pattern in the text, in time $O(nm)$. A faster algorithm, in the absence of wild cards, is Abrahamson's algorithm [10] that runs in $O(n\sqrt{m \log m})$ time. Abrahamson's algorithm can be extended to solve pattern matching with mismatches and wild cards, as we prove in Section 2.2.1. The new algorithm runs in $O(n\sqrt{g \log m})$ time, where g is the number of non-wild card positions in the pattern. This gives a simpler and faster alternative to an algorithm proposed in [11].

In the literature, we also find algorithms that approximate the number of mismatches for every alignment. For example, an approximate algorithm for pattern matching with mismatches, in the absence of wild cards, that runs in $O(rn \log m)$ time, where r is the number of iterations of the algorithm, is given in [12]. Every distance reported has a variance bounded by $(m - c_i)/r^2$ where c_i is the exact number of matches for alignment i.

Furthermore, a randomized algorithm that approximates the Hamming distance for every alignment within an ϵ factor and runs in $O(n \log^c m/\epsilon^2)$ time, in the absence of wild cards, is given in [13]. Here, c is a small constant. We extend this algorithm to pattern matching with mismatches and wild cards, in

Section 2.3. The new algorithm approximates the Hamming distance for every alignment within an ϵ factor in time $O(n \log^2 m/\epsilon^2)$ with high probability.

Recent work has also addressed the online version of pattern matching, where the text is received in a streaming model, one character at a time, and it cannot be stored in its entirety (see, e.g., [14–16]). Another version of this problem matches the pattern against multiple input streams (see, e.g., [17]). Another interesting problem is to sample a representative set of mismatches for every alignment (see, e.g., [18]).

1.2. Pattern Matching with K Mismatches

For the k mismatches problem, without wild cards, two algorithms that run in $O(nk)$ time are presented in [19,20]. A faster algorithm, that runs in $O(n\sqrt{k \log k})$ time, is given in [11]. This algorithm combines the two main techniques known in the literature for pattern matching with mismatches: filtering and convolutions. We give a significantly simpler algorithm in Section 2.2.3, having the same worst case run time. The new algorithm will never perform more operations than the one in [11] during marking and convolution.

An intermediate problem is to check if the Hamming distance is less or equal to k for a subset of the aligned positions. This problem can be solved with the Kangaroo method proposed in [11] at a cost of $O(k)$ time per alignment, using $O(n + m)$ additional memory. We show how to achieve the same run time per alignment using only $O(m)$ additional memory, in Section 2.2.2.

Further, we look at the version of k mismatches where wild cards are allowed in the text and the pattern. For this problem, two randomized algorithms are presented in [17]. The first one runs in $O(nk \log n \log m)$ time, and the second one in $O\left(n \log m(k + \log n \log \log n)\right)$ time. Both are Monte Carlo algorithms, *i.e.*, they output the correct answer with high probability. The same paper also gives a deterministic algorithm with a run time of $O(nk^2 \log^3 m)$. Furthermore, a deterministic $O(nk \log^2 m(\log^2 k + \log \log m))$ time algorithm is given in [21]. We present a Las Vegas algorithm (that always outputs the correct answer), in Section 2.4.3, which runs in time $O(nk \log^2 m + n \log^2 m \log n + n \log m \log n \log \log n)$ with high probability.

An algorithm for k mismatches with wild cards in either the text or the pattern (but not both) is given in [22]. This algorithm runs in $O(nm^{1/3}k^{1/3}log^{2/3}m)$ time.

1.3. Our Results

The contributions of this paper can be summarized as follows.

For pattern matching with mismatches:

- An algorithm for pattern matching with mismatches and wild cards that runs in $O(n\sqrt{g \log m})$ time, where g is the number of non-wild card positions in the pattern; see Section 2.2.1.
- A randomized algorithm that approximates the Hamming distance for every alignment, when wild cards are present, within an ϵ factor in time $O(n \log^2 m/\epsilon^2)$ with high probability; see Section 2.3.

For pattern matching with k mismatches:

- An algorithm for pattern matching with k mismatches, without wild cards, that runs in $O(n\sqrt{k \log k})$ time; this algorithm is simpler and has a better expected run time than the one in [11]; see Section 2.2.3.
- An algorithm that tests if the Hamming distance is less than k for a subset of the alignments, without wild cards, at a cost of $O(k)$ time per alignment, using only $O(m)$ additional memory; see Section 2.2.2.
- A Las Vegas algorithm for the k mismatches problem with wild cards that runs in time $O(nk \log^2 m + n \log^2 m \log n + n \log m \log n \log \log n)$ with high probability; see Section 2.4.3.

The rest of the paper is organized as follows. First, we introduce some notations and definitions. Then, we describe the exact, deterministic algorithms for pattern matching with mismatches and for k mismatches. Then, we present the randomized and approximate algorithms: first the algorithm for approximate counting of mismatches in the presence of wild cards, then the Las Vegas algorithm for k mismatches with wild cards. Finally, we present an empirical run time comparison of the deterministic algorithms and conclusions.

2. Materials and Methods

2.1. Some Definitions

Given two strings $T = t_1 t_2 \dots t_n$ and $P = p_1 p_2 \dots p_m$ (with $m \leq n$), the convolution of T and P is a sequence $C = c_1, c_2, \dots, c_{n-m+1}$ where $c_i = \sum_{j=1}^{m} t_{i+j-1} p_j$, for $1 \leq i \leq (n - m + 1)$. This convolution can be computed in $O(n \log m)$ time using the fast Fourier transform. If the convolutions are applied on binary inputs, as is often the case in pattern matching applications, some speedup techniques are presented in [23].

In the context of randomized algorithms, by high probability, we mean a probability greater or equal to $(1 - n^{-\epsilon})$ where n is the input size and ϵ is a probability parameter usually assumed to be a constant greater than 0. The run time of a Las Vegas algorithm is said to be $\widetilde{O}(f(n))$ if the run time is no more than $c\epsilon f(n)$ with probability greater or equal to $(1 - n^{-\epsilon})$ for all $n \geq n_0$, where c and n_0 are some constants and for any constant $\epsilon \geq 1$.

In the analysis of our algorithms, we will employ the following Chernoff bounds.

Chernoff bounds [24]: These bounds can be used to closely approximate the tail ends of a binomial distribution.

A Bernoulli trial has two outcomes, namely success and failure, the probability of success being p. A binomial distribution with parameters n and p, denoted as $B(n, p)$, is the number of successes in n independent Bernoulli trials.

Let X be a binomial random variable whose distribution is $B(n, p)$. If m is any integer $> np$, then the following are true:

$$Prob.[X > m] \leq \left(\frac{np}{m}\right)^m e^{m-np} \quad (1)$$

$$Prob.[X > (1+\delta)np] \leq e^{-\delta^2 np/3}; \text{ and} \quad (2)$$

$$Prob.[X < (1-\delta)np] \leq e^{-\delta^2 np/2} \quad (3)$$

for any $0 < \delta < 1$.

2.2. Deterministic Algorithms

In this section, we present deterministic algorithms for pattern matching with mismatches. We start with a summary of two well-known techniques for counting matches: convolution and marking (see, e.g., [11]). In terms of notation, $T_{i..j}$ is the substring of T between i and j and T_i stands for $T_{i..i+m-1}$. Furthermore, the value at position i in array X is denoted by $X[i]$.

Convolution: Given a string S and a character α, define string S^α, such that $S^\alpha[i] = 1$ if $S[i] = \alpha$, and 0 otherwise. Let $C^\alpha = convolution(T^\alpha, P^\alpha)$. Then, $C^\alpha[i]$ gives the number of matches between P and T_i where the matching character is α. Then, $\sum_{\alpha \in \Sigma} C^\alpha[i]$ is the total number of matches between P and T_i.

Marking: Given a character α, let $Pos[\alpha]$ be the set of positions where character α is found in P (*i.e.*, $Pos[\alpha] = \{i | 1 \leq i \leq m, p_i = \alpha\}$). Note that, if $t_i = \alpha$, then the alignment between P and T_{i-j+1} will match $t_i = p_j = \alpha$, for all $j \in Pos[\alpha]$. This gives the marking algorithm: for every position i in the text, increment the number of matches for alignment $i - j + 1$, for all $j \in Pos[t_i]$. In practice, we are interested in doing the marking only for certain characters, meaning we will do the incrementing only for the positions $t_i = \alpha$ where $\alpha \in \Gamma \subseteq \Sigma$. The algorithm then takes $O(n \max_{\alpha \in \Gamma} |Pos_\alpha|)$ time. The pseudocode is given in Algorithm 1.

Algorithm 1: Mark(T, n, Γ)

```
for i ← 1 to n do M[i] = 0;
for i ← 1 to n do
    if t_i ∈ Γ then
        for j ∈ Pos[t_i] do
            if i − j + 1 > 0 then M[i − j + 1]++ ;
return M;
```

2.2.1. Pattern Matching with Mismatches

For pattern matching with mismatches, without wild cards, Abrahamson [10] gave the following $O(n\sqrt{m \log m})$ time algorithm. Let A be a set of the most frequent characters in the pattern: (1) using convolutions, count how many matches each character in A contributes to every alignment; (2) using marking, count how many matches each character in $\Sigma - A$ contributes to every alignment; and (3) add the two numbers to find for every alignment, the number of matches between the pattern and the text. The convolutions take $O(|A| n \log m)$ time. A character in $\Sigma - A$ cannot appear more than $m/|A|$ times in the pattern; otherwise, each character in A has a frequency greater than $m/|A|$, which is not possible.

Thus, the run time for marking is $O(nm/|A|)$. If we equate the two run times, we find the optimal $|A| = \sqrt{m/\log m}$, which gives a total run time of $O(n\sqrt{m \log m})$.

An Example: Consider the case of $T = 2\ 3\ 1\ 1\ 4\ 1\ 2\ 3\ 4\ 4\ 2\ 1\ 1\ 3\ 2$ and $P = 1\ 2\ 3\ 4$. Since each character in the pattern occurs an equal number of times, we can pick A arbitrarily. Let $A = \{1, 2\}$. In Step 1, convolution is used to count the number of matches contributed by each character in A. We obtain an array $M_1[1:12]$, such that $M_1[i]$ is the number of matches contributed by characters in A to the alignment of P with T_i, for $1 \leq i \leq 12$. In this example, $M_1 = [0, 0, 1, 1, 0, 2, 0, 0, 0, 1, 0, 1]$. In Step 2, we compute, using marking, the number of matches contributed by the characters 3 and 4 to each alignment between T and P. We get another array $M_2[1:12]$, such that $M_2[i]$ is the number of matches contributed by 3 and 4 to the alignment between T_i and P, for $1 \leq i \leq 12$. Specific to this example, $M_2 = [0, 1, 0, 0, 0, 2, 1, 0, 0, 0, 0, 1]$. In Step 3, we add M_1 and M_2 to get the number of matches between T_i and P, for $1 \leq i \leq 12$. In this example, this sum yields: $[0, 1, 1, 1, 0, 4, 1, 0, 0, 1, 0, 2]$.

For pattern matching with mismatches and wild cards, a fairly complex algorithm is given in [11]. The run time of this algorithm is $O(n\sqrt{g} \log m)$ where g is the number of non-wild card positions in the pattern. The problem can also be solved through a simple modification of Abrahamson's algorithm, in time $O(n\sqrt{m \log m})$, as pointed out in [17]. We now prove the following result:

Theorem 1. *Pattern matching with mismatches and wild cards can be solved in $O(n\sqrt{g \log m})$ time, where g is the number of non-wild card positions in the pattern.*

Proof. Ignoring the wild cards for now, let A be the set of the most frequent characters in the pattern. As above, count matches contributed by characters in A and $\Sigma - A$ using convolution and marking, respectively. By a similar reasoning as above, the characters used in the marking phase will not appear more than $g/|A|$ times in the pattern. If we equate the run times for the two phases, we obtain $O(n\sqrt{g \log m})$ time. We are now left to count how many matches are contributed by the wild cards. For a string S and a character α, define $S^{\neg\alpha}$ as $S^{\neg\alpha}[i] = 1 - S^{\alpha}[i]$. Let w be the wild card character. Compute $C = convolution(T^{\neg w}, P^{\neg w})$. Then, for every alignment i, the number of positions that have a wild card either in the text, or the pattern, or both, is $m - C[i]$. Add $m - C[i]$ to the previously-computed counts and output. The total run time is $O(n\sqrt{g \log m})$. □

2.2.2. Pattern Matching with K Mismatches

For the k mismatches problem, without wild cards, an $O(k(m \log m + n))$ time algorithm that requires $O(k(m + n))$ additional space is presented in [19]. Another algorithm, that takes $O(m \log m + kn)$ time and uses only $O(m)$ additional space, is presented in [20]. We define the following problem, which is of interest in the discussion.

Problem 1. *Subset k mismatches: Given a text T of length n, a pattern P of length m, a set of positions $S = \{i | 1 \leq i \leq n - m + 1\}$ and an integer k, output the positions $i \in S$ for which $Hd(P, T_i) \leq k$.*

The subset k mismatches problem becomes the regular k mismatches problem if $|S| = n - m + 1$. Thus, it can be solved by the $O(nk)$ algorithms mentioned above. However, if $|S| << n$, then the $O(nk)$ algorithms are too costly. A better alternative is to use the Kangaroo method proposed in [11].

The Kangaroo method can verify if $Hd(P, T_i) \leq k$ in $O(k)$ time for any i. The method works as follows. Build a suffix tree of $T\#P$ and enhance it to support $O(1)$ lowest common ancestor (LCA) queries. For a given i, perform an LCA query to find the position of the first mismatch between P and T_i. Let this position be j. Then, perform another LCA to find the first mismatch between $P_{j+1..m}$ and $T_{i+j+1..i+m-1}$, which is the second mismatch of alignment i. Continue to "jump" from one mismatch to the next, until the end of the pattern is reached or we have found more than k mismatches. The Kangaroo method can process $|S|$ positions in $O(n + m + |S|k)$ time, and it uses $O(n + m)$ additional memory for the LCA enhanced suffix tree. We now prove the following result:

Theorem 2. *Subset k mismatches can be solved in $O(n + m + |S|k)$ time using only $O(m)$ additional memory.*

Proof. The algorithm is the following. Build an LCA-enhanced suffix tree of the pattern. Scan the text from left to right: (1) find the longest unscanned region of the text that can be found somewhere in the pattern, say starting at position i of the pattern; call this region of the text R; therefore, R is identical to $P_{i..i+|R|-1}$.; and (2) for every alignment in S that overlaps R, count the number of mismatches between R and the alignment, within the overlap region. To do this, consider an alignment in S that overlaps R, such that the beginning of R aligns with the j-th character in the pattern. We want to count the number of mismatches between R and $P_{j..j+|R|-1}$. However, since R is identical to $P_{i..i+|R|-1}$, we can simply compare $P_{i..i+|R|-1}$ and $P_{j..j+|R|-1}$. This comparison can be done efficiently by jumping from one mismatch to the next, like in the Kangaroo method. Repeat from Step 1 until the entire text has been scanned. Every time we process an alignment, in Step 2, we either discover at least one additional mismatch or we reach the end of the alignment. This is true, because, otherwise, the alignment would match the text for more than $|R|$ characters, which is not possible, from the way we defined R. Every alignment for which we have found more than k mismatches is excluded from further consideration to ensure $O(k)$ time per alignment. It takes $O(m)$ time to build the LCA enhanced suffix tree of the pattern and $O(n)$ additional time to scan the text from left to right. Thus, the total run time is $O(n + m + |S|k)$ with $O(m)$ additional memory. The pseudocode is given in Algorithm 2. □

2.2.3. An $O(n\sqrt{k \log k})$ Time Algorithm for K Mismatches

For the k mismatches problem, without wild cards, a fairly complex $O(n\sqrt{k \log k})$ time algorithm is given in [11]. The algorithm classifies the inputs into several cases. For each case, it applies a combination of marking followed by a filtering step, the Kangaroo method, or convolutions. The goal is to not exceed $O(n\sqrt{k \log k})$ time in any of the cases. We now present an algorithm with only two cases that has the same worst case run time. The new algorithm can be thought of as a generalization of the algorithm in [11], as we will discuss later. This generalization not only greatly simplifies the algorithm, but it also reduces the expected run time. This happens because we use information about the frequency of the characters in the text and try to minimize the work done by convolutions and marking.

Algorithm 2: Subset k mismatches(S, T, P, k)

input : S - set of positions in the text; $T_{1..n}$ - text; $P_{1..m}$ -pattern; k - max number of mismatches;
output: M - the positions in S for which the pattern matches the text with at most k mismatches;
begin
  Assume we have a suffix tree/array of the pattern;
  for $a \in S$ **do** $M[a] = 0$;
  $i = 1$;
  while $i \leq n$ **do**
    Find the largest l, such that $T_{i..i+l-1}$ describes a path in the suffix tree;
    This means that $T_{i..i+l-1} = P_{j..j+l-1}$ for some j;
    for $a \in S$ **where** $a \leq i < a + m$ **do**
      $M[a] =$ `countMismatches` $(M[a], i - a + 1, j, l)$;
      if $t_{i+l} \neq p_{j+l}$ **then** $M[a] = M[a] + 1$;
      if $M[a] > k$ **then** $S = S - \{a\}$;
    $i = i + l + 1$;
  return $\{a \in S | M[a] \leq k\}$
function `countMismatches`(c, s_1, s_2, l)
  input : c - current number of mismatches; s_1, s_2 - starting positions of two suffixes of the pattern; l - a maximum length;
  output: compare the two suffixes on their first l positions and add to c the number of mismatches found; if c exceeds k, return $k + 1$, otherwise return the updated c ;
  begin
    while $l > 0$ **and** $c \leq k$ **do**
      $d =$ `lcp`(s_1, s_2); // longest common prefix
      if $d \geq l$ **then return** c;
      $c = c + 1$;
      $d = d + 1$;
      $s_1 = s_1 + d$;
      $s_2 = s_2 + d$;
      $l = l - d$;
  return c

We will now give the intuition for this algorithm. For any character $\alpha \in \Sigma$, let f_α be its frequency in the pattern and F_α be its frequency in the text. Note that in the marking algorithm, a specific character α will contribute to the runtime a cost of $F_\alpha \times f_\alpha$. On the other hand, in the case of convolution, a character α costs us one convolution, regardless of how frequent α is in the text or the pattern. Therefore, we want to use infrequent characters for marking and frequent characters for convolution. The balancing of the two will give us the desired runtime.

A position j in the pattern where $p_j = \alpha$ is called an instance of α. Consider every instance of character α as an object of size 1 and cost F_α. We want to fill a knapsack of size $2k$ at a minimum cost

and without exceeding a given budget B. The $2k$ instances will allow us to filter some of the alignments with more than k mismatches, as will become clear later. This problem can be optimally solved by a greedy approach, where we include in the knapsack all of the instances of the least expensive character, then all of the instances of the second least expensive character, and so on, until we have $2k$ items or we have exceeded B. The last character considered may have only a subset of its instances included, but for the ease of explanation, assume that there are no such characters.

Note: Even though the above is described as a knapsack problem, the particular formulation can be optimally solved in linear time. This formulation should not be confused with other formulations of the knapsack problem that are NP-complete.

Case (1): Assume we can fill the knapsack at a cost $C \leq B$. We apply the marking algorithm for the characters whose instances are included in the knapsack. It is easy to see that the marking takes time $O(C)$ and creates C marks. For alignment i, if the pattern and the text match for all of the $2k$ positions in the knapsack, we will obtain exactly $2k$ marks at position i. Conversely, any position that has less than k marks must have more than k mismatches, so we can filter it out. Therefore, there will be at most C/k positions with k marks or more. For such positions, we run subset k mismatches to confirm which of them have less than k mismatches. The total runtime of the algorithm in this case is $O(C)$.

Case (2): If we cannot fill the knapsack within the given budget B, we do the following: for the characters we could fit in the knapsack, we use the marking algorithm to count the number of matches that they contribute to each alignment. For characters not in the knapsack, we use convolutions to count the number of matches that they contribute to each alignment. We add the two counts and get the exact number of matches for every alignment.

Note that at least one of the instances in the knapsack has a cost larger than $B/(2k)$ (if all of the instances in the knapsack had a cost less or equal to $B/(2k)$, then we would have at least $2k$ instances in the knapsack). Furthermore, note that all of the instances not in the knapsack have a cost at least as high as any instance in the knapsack, because we greedily fill the knapsack starting with the least costly instances. This means that every character not in the knapsack appears in the text at least $B/(2k)$ times. This means that the number of characters not in the knapsack does not exceed $n/(B/(2k))$. Therefore, the total cost of convolutions is $O(nk/B \log m)$. Since the cost of marking was $O(B)$, we can see that the best value of B is the one that equalizes the two costs. This gives $B = O(n\sqrt{k \log m})$. Therefore, the algorithm takes $O(n\sqrt{k \log m})$ time. If $k < m^{1/3}$, we can employ a different algorithm that solves the problem in linear time, as in [11]. For larger k, $O(\log m) = O(\log k)$, so the run time becomes $O(n\sqrt{k \log k})$. We call this algorithm knapsack k mismatches. The pseudocode is given in Algorithm 3. The following theorem results.

Theorem 3. *Knapsack k mismatches has worst case run time $O(n\sqrt{k \log k})$.*□

Algorithm 3: Knapsack k mismatches(T, P, k)

input : $T_{1..n}$ - text; $P_{1..m}$ -pattern; k - max number of mismatches;
output: S - set of positions in the text where the pattern matches with at most k mismatches;
begin
  Compute F_i and f_i for every $i \in \Sigma$;
  Sort Σ with respect to F_i ;
  $s = 0$;
  $c = 0$;
  $i = 1$;
  $B = n\sqrt{k \log k}$;
  while $s < 2k$ **and** $c < B$ **do**
    $t = \min(f_i, 2k - s)$;
    $s = s + t$;
    $c = c + t \times F_i$;
    $i = i + 1$;
  $\Gamma = \Sigma[1..i]$;
  $M = \texttt{Mark}(T, n, \Gamma)$; // M counts matches
  if $s = 2k$ **then**
    $S = \{i | M[i] \geq k\}$;
    return `Subset k mismatches` (S, T, P, k);
  else
    for $\alpha \in \Sigma - \Gamma$ **do**
      $C = \texttt{convolution}(T^\alpha, P^\alpha)$;
      for $i \leftarrow 1$ **to** n **do**
        $M[i] = M[i] + C[i]$;
    $S = \{i | M[i] \geq m - k\}$;
    return S;

We can think of the algorithm in [11] as a special case of our algorithm where, instead of trying to minimize the cost of the $2k$ items in the knapsack, we just try to find $2k$ items for which the cost is less than $O(n\sqrt{k \log m})$. As a result, it is easy to verify the following:

Theorem 4. *Knapsack k mismatches spends at most as much time as the algorithm in [11] to do convolutions and marking.*

Proof. Observation: In all of the cases presented below, knapsack k mismatches can have a run time as low as $O(n)$, for example if there exists one character α with $f_\alpha = O(k)$ and $F_\alpha = O(n/k)$.

Case 1: $|\Sigma| \geq 2k$. The algorithm in [11] chooses $2k$ instances of distinct characters to perform marking. Therefore, for every position of the text, at most one mark is created. If the number of

marks is M, then the cost of the marking phase is $O(n + M)$. The number of remaining positions after filtering is no more than M/k, and thus, the algorithm takes $O(n + M)$ time. Our algorithm puts in the knapsack 2k instances, of not necessarily different characters, such that the number of marks B is minimized!Therefore, $B \leq M$, and the total runtime is $O(n + B)$.

Case 2: $|\Sigma| < 2\sqrt{k}$. The algorithm in [11] performs one convolution per character to count the total number of matches for every alignment, for a run time of $\Omega(|\Sigma| n \log m)$. In the worst case, knapsack k mismatches cannot fill the knapsack at a cost $B < |\Sigma| n \log m$, so it defaults to the same run time. However, in the best case, the knapsack can be filled at a cost B as low as $O(n)$ depending on the frequency of the characters in the pattern and the text. In this case, the runtime will be $O(n)$.

Case 3: $2\sqrt{k} \leq |\Sigma| \leq 2k$. A symbol that appears in the pattern at least $2\sqrt{k}$ times is called frequent.

Case 3.1: There are at least $\sqrt{k}$ frequent symbols. The algorithm in [11] chooses $2\sqrt{k}$ instances of $\sqrt{k}$ frequent symbols to do marking and filtering at a cost $M \leq 2n\sqrt{k}$. Since knapsack k mismatches will minimize the marking time B, we have $B \leq M$, so the run time is the same as for [11] only in the worst case.

Case 3.2: There are $A < \sqrt{k}$ frequent symbols. The algorithm in [11] first performs one convolution for each frequent character for a run time of $O(An \log m)$. Two cases remain:

Case 3.2.1: All of the instances of the non-frequent symbols number less than $2k$ positions. The algorithm in [11] replaces all instances of frequent characters with wild cards and applies a $O(n\sqrt{g} \log m)$ algorithm to count mismatches, where g is the number of non-wild card positions. Since $g < 2k$, the run time for this stage is $O(n\sqrt{k} \log m)$, and the total run time is $O(An \log m + n\sqrt{k} \log m)$. Knapsack k mismatches can always include in the knapsack all of the instances of non-frequent symbols, since their total cost is no more than $O(n\sqrt{k})$, and in the worst case, do convolutions for the remaining characters. The total run time is $O(An \log m + n\sqrt{k})$. Of course, depending on the frequency of the characters in the pattern and text, knapsack k mismatch may not have to do any convolutions.

Case 3.2.2: All of the instances of the non-frequent symbols number at least $2k$ positions. The algorithm in [11] chooses $2k$ instances of infrequent characters to do marking. Since each character has a frequency less than $2\sqrt{k}$, the time for marking is $M < 2n\sqrt{k}$, and there are no more than M/k positions left after filtering. Knapsack k mismatches chooses characters in order to minimize the time B for marking, so again $B \leq M$. □

2.3. Approximate Counting of Mismatches

The algorithm of [13] takes as input a text $T = t_1 t_2 \ldots t_n$ and a pattern $P = p_1 p_2 \ldots p_m$ and approximately counts the Hamming distance between T_i and P for every $1 \leq i \leq (n - m + 1)$. In particular, if the Hamming distance between T_i and P is H_i for some i, then the algorithm outputs h_i where $H_i \leq h_i \leq (1 + \epsilon)H_i$ for any $\epsilon > 0$ with high probability (*i.e.*, a probability of $\geq (1 - m^{-\alpha})$). The run time of the algorithm is $O(n \log^2 m/\epsilon^2)$. In this section, we show how to extend this algorithm to the case where there could be wild cards in the text and/or the pattern.

Let Σ be the alphabet under concern, and let $\sigma = |\Sigma|$. The algorithm runs in phases, and in each phase, we randomly map the elements of Σ to $\{1, 2\}$. A wild card is mapped to a zero. Under this mapping, we transform T and P to T' and P', respectively. We then compute a vector C where

$C[i] = \sum_{j=1}^{m}(t'_{i+j-1} - p'_j)^2 t'_{i+j-1} p'_j$. This can be done using $O(1)$ convolution operations (as in Section 2.4.1; see also [17]). A series of r such phases (for some relevant value of r) is done, at the end of which, we produce estimates on the Hamming distances. The intuition is that if a character x in T' is aligned with a character y in P', then across all of the r phases, the expected contribution to C from these characters is r if $x \neq y$ (assuming that x and y are non-wild cards). If $x = y$ or if one or both of x and y are a wild card, the contribution to C is zero.

Algorithm 4:

1. **for** $i \leftarrow 1$ **to** $(n - m + 1)$ **do** $C[i] = 0$;
2. **for** $\ell \leftarrow 1$ **to** r **do**
   - Let Q be a random mapping of Σ to $\{1, 2\}$.
   - In particular, each element of Σ is mapped to 1 or 2 randomly with equal probability.
   - Each wild card is mapped to a zero.
   - Obtain two strings T' and P' where $t'_i = Q(t_i)$ for $1 \leq i \leq n$ and $p'_j = Q(p_j)$ for $1 \leq j \leq m$;
   - Compute a vector C_ℓ where $C_\ell[i] = \sum_{j=1}^{m}(t'_{i+j-1} - p'_j)^2\, t'_{i+j-1} p'_j$ for $1 \leq i \leq (n - m + 1)$;
   - **for** $i \leftarrow 1$ **to** $(n - m + 1)$ **do** $C[i] = C[i] + C_\ell[i]$;
3. **for** $i \leftarrow 1$ **to** $(n - m + 1)$ **do**
   - Output $h_i = \frac{C[i]}{r}$;
   - Here, h_i is an estimate on the Hamming distance H_i between T_i and P.

Analysis: Let x be a character in T, and let y be a character in P. Clearly, if $x = y$ or if one or both of x and y are a wild card, the contribution of x and y to any $C_\ell[i]$ is zero. If x and y are non-wild cards and if $x \neq y$, then the expected contribution of these to any $C_\ell[i]$ is 1. Across all of the r phases, the expected contribution of x and y to any $C_\ell[i]$ is r. For a given x and y, we can think of each phase as a Bernoulli trial with equal probabilities for success and failure. A success refers to the possibility of $Q(x) \neq Q(y)$. The expected number of successes in r phases is $\frac{r}{2}$. Using Chernoff bounds (Equation 2), this contribution is no more than $(1 + \epsilon)r$ with probability $\geq 1 - \exp(-\epsilon^2 r/6)$. The probability that this statement holds for every pair (x, y) is $\geq 1 - m^2 \exp(-\epsilon^2 r/6)$. This probability will be $\geq 1 - m^{-\alpha}/2$ if $r \geq \frac{6(\alpha+3)\log_e m}{\epsilon^2}$. Similarly, we can show that for any pair of non-wild card characters, the contribution of them to any $C_\ell[i]$ is no less than $(1 - \epsilon)r$ with probability $\geq 1 - m^{-\alpha}/2$ if $r \geq \frac{4(\alpha+3)\log_e m}{\epsilon^2}$.

Put together, for any pair (x, y) of non-wild cards, the contribution of x and y to any $C_\ell[i]$ is in the interval $(1 \pm \epsilon)r$ with probability $\geq (1 - m^{-\alpha})$ if $r \geq \frac{6(\alpha+3)\log_e m}{\epsilon^2}$. Let H_i be the Hamming distance between T_i and P for some i $(1 \leq i \leq (n - m + 1))$. Then, the estimate h_i on H_i will be in the interval $(1 \pm \epsilon)H_i$ with probability $\geq (1 - m^{-\alpha})$. As a result, we get the following Theorem.

Theorem 5. *Given a text T and a pattern P, we can estimate the Hamming distance between T_i and P, for every i, $1 \leq i \leq (n-m+1)$, in $O(n \log^2 m/\epsilon^2)$ time. If H_i is the Hamming distance between T_i and P, then the above algorithm outputs an estimate that is in the interval $(1 \pm \epsilon)H_i$ with high probability.*

Observation 1. In the above algorithm, we can ensure that $h_i \geq H_i$ and $h_i \leq (1 + \epsilon)H_i$ with high probability by changing the estimate computed in Step 3 of Algorithm 4 to $\frac{C[i]}{(1-\epsilon)r}$.

Observation 2. As in [13], with $O\left(\frac{m^2 \log m}{\epsilon^2}\right)$ pre-processing, we can ensure that Algorithm 4 never errs (*i.e.*, the error bounds on the estimates will always hold).

2.4. A Las Vegas Algorithm for K Mismatches

2.4.1. The 1 Mismatch Problem

Problem definition: For this problem, also, the inputs are two strings T and P with $|T| = n, |P| = m$, $m \leq n$ and possible wild cards in T and P. Let T_i stand for the substring $t_i t_{i+1} \ldots t_{i+m-1}$, for any i, with $1 \leq i \leq (n-m+1)$. The problem is to check if the Hamming distance between T_i and P is exactly 1, for $1 \leq i \leq (n-m+1)$. The following Lemma is shown in [17].

Lemma 1. *The 1 mismatch problem can be solved in $O(n \log m)$ time using a constant number of convolution operations.*

The algorithm: Assume that each wild card in the pattern as well as the text is replaced with a zero. Furthermore, assume that the characters in the text, as well as the pattern are integers in the range $[1 : |\Sigma|]$ where Σ is the alphabet of concern. Let $e_{i,j}$ stand for the "error term" introduced by the character t_{i+j-1} in T_i and the character p_j in P, and its value is $(t_{i+j-1} - p_j)^2 t_{i+j-1} p_j$. Furthermore, let $E_i = \sum_{j=1}^{m} e_{i,j}$. There are four steps in the algorithm:

1. Compute E_i for $1 \leq i \leq (n-m+1)$. Note that E_i will be zero if T_i and P match (assuming that a wild card can be matched with any character). $E_i = \sum_{j=1}^{m} (t_{i+j-1} - p_j)^2 t_{i+j-1} p_j = \sum_{j=1}^{m} t_{i+j-1}^3 p_j + \sum_{j=1}^{m} t_{i+j-1} p_j^3 - 2\sum_{j=1}^{m} t_{i+j-1}^2 p_j^2$. Thus, this step can be completed with three convolution operations.
2. Compute E'_i for $1 \leq i \leq (n-m+1)$, where $E'_i = \sum_{j=1}^{m} (i+j-1)(t_{i+j-1} - p_j)^2 p_j t_{i+j-1}$ (for $1 \leq i \leq (n-m+1)$). Like Step 1, this step can also be completed with three convolution operations.
3. Let $B_i = E'_i / E_i$ if $E_i \neq 0$, for $1 \leq i \leq (n-m+1)$. Note that if the Hamming distance between T_i and P is exactly one, then B_i will give the position in the text where this mismatch occurs.
4. If for any i ($1 \leq i \leq (n-m+1)$), $E_i \neq 0$ and if $(t_{B_i} - p_{B_i - i + 1})^2 t_{B_i} p_{B_i - i + 1} = E_i$, then we conclude that the Hamming distance between T_i and P is exactly one.

Note: If the Hamming distance between T_i and P is exactly 1 (for any i), then the above algorithm will not only detect it, but also will identify the position where there is a mismatch. Specifically, it will identify the integer j, such that $t_{i+j-1} \neq p_j$.

An example. Consider the case where $\Sigma = \{1, 2, 3, 4, 5, 6\}$, $T = 5\ 6\ 4\ 6\ 2\ *\ 3\ 3\ 4\ 5\ 1\ *\ 1\ 2\ 5\ 5\ 5\ 6\ 4\ 3$ and $P = 2\ 5\ 6\ 3$. Here, $*$ represents the wild card.

In Step 1, we compute E_i, for $1 \leq i \leq 17$. For example, $E_1 = (5-2)^2 \times 5 \times 2 + (6-5)^2 \times 6 \times 5 + (4-6)^2 \times 4 \times 6 + (6-3)^2 \times 6 \times 3 = 378$; $E_2 = (6-2)^2 \times 6 \times 2 + (4-5)^2 \times 4 \times 5 + (6-6)^2 \times 6 \times 6 + (2-3)^2 \times 2 \times 3 = 218$; $E_3 = 254$; $E_5 = (2-2)^2 \times 2 \times 2 + 0 + (6-3)^2 \times 6 \times 3 + (3-3)^2 \times 3 \times 3 = 162$. Note that since t_5 is a wild card, it matches with any character in the pattern. Furthermore, $E_9 = 182$.

In Step 2, we compute E'_i, for $1 \leq i \leq 17$. For instance, $E'_1 = 1 \times (5-2)^2 \times 5 \times 2 + 2 \times (6-5)^2 \times 6 \times 5 + 3(4-6)^2 \times 4 \times 6 + 4 \times (6-3)^2 \times 6 \times 3 = 1410$; $E'_5 = 5 \times (2-2)^2 \times 2 \times 2 + 0 + 7 \times (6-3)^2 \times 6 \times 3 + 8 \times (3-3)^2 \times 3 \times 3 = 1134$.

In Step 3, the value of $B_i = E'_i/E_i$ is computed for $1 \leq i \leq 17$. For example, $B_1 = E'_1/E_1 = 1410/378 \approx 3.73$; $B_5 = E'_5/E_5 = 1134/162 = 7$.

In Step 4, we identify all of the positions in the text corresponding to a single mismatch. For instance, we note that $E_5 \neq 0$ and $(t_7 - p_3)^2 \times t_7 \times p_3 = E_5$. As a result, Position 5 in the text corresponds to 1 mismatch.

2.4.2. The Randomized Algorithms of [17]

Two different randomized algorithms are presented in [17] for solving the k mismatches problem. Both are Monte Carlo algorithms. In particular, they output the correct answers with high probability. The run times of these algorithms are $O(nk \log m \log n)$ and $O(n \log m(k + \log n \log \log n))$, respectively. In this section, we provide a summary of these algorithms.

The first algorithm has $O(k \log n)$ sampling phases, and in each phase, a 1 mismatch problem is solved. Each phase of sampling works as follows. We choose m/k positions of the pattern uniformly at random. The pattern P is replaced by a string P' where $|P'| = m$, the characters in P' in the randomly chosen positions are the same as those in the corresponding positions of P, and the rest of the characters in P' are set to wild cards. The 1 mismatch algorithm of Lemma 1 is run on T and P'. In each phase of random sampling, for each i, we get to know if the Hamming distance between T_i and P' is exactly 1, and, if so, identify the j, such that $t_{i+j-1} \neq p'_j$.

As an example, consider the case when the Hamming distance between T_i and P is k (for some i). Then, in each phase of sampling, we would expect to identify exactly one of the positions (*i.e.*, j) where T_i and P differ (*i.e.*, $t_{i+j-1} \neq p_j$). As a result, in an expected k phases of sampling, we will be able to identify all of the k positions in which T_i and P differ. It can be shown that if we make $O(k \log n)$ sampling phases, then we can identify all of the k mismatches with high probability [17]. It is possible that the same j might be identified in multiple phases. However, we can easily keep track of this information to identify the unique j values found in all of the phases.

Let the number of mismatches between T_i and P be q_i (for $1 \leq i \leq (n - m + 1)$. If $q_i \leq k$, the algorithm of [17] will compute q_i exactly. If $q_i > k$, then the algorithm will report that the number of mismatches is $> k$ (without estimating q_i), and this answer will be correct with high probability. The algorithm starts off by first computing E_i values for every T_i. A list $L(i)$ of all of the mismatches found for T_i is kept, for every i. Whenever a mismatch is found between T_i and P (say in position $(i + j - 1)$ of the text), the value of E_i is reduced by $e_{i,j}$. If at any point in the algorithm, E_i becomes zero for any i, this means that we have found all of the q_i mismatches between T_i and P, and $L(i)$ will have the positions in the text where these mismatches occur. Note that if the Hamming distance between T_i and P is much larger than k (for example, close or equal to m), then the probability that in a random sample we isolate a single mismatch is very low. Therefore, if the number of sample phases is only $O(k \log n)$, the algorithm can only be Monte Carlo. Even if q_i is less or equal to k, there is a small probability that we may not be able to find all of the q_i mismatches. Call this algorithm Algorithm 5. If for each i, we either get all of the q_i mismatches (and hence, the corresponding E_i is zero) or we have found more than

k mismatches between T_i and P, then we can be sure that we have found all of the correct answers (and the algorithm will become Las Vegas).

An example. Consider the example of $\Sigma = \{1, 2, 3, 4, 5, 6\}$, $T = 5\,6\,4\,6\,2\,*\,3\,3\,4\,5\,1\,*\,1\,2\,5\,5\,5\,6\,4\,3$ and $P = 2\,5\,6\,3$. Here, $*$ represents the wild card.

As has been computed before, $E_5 = 162$ and $E_9 = 182$. Let $k = 2$. In each phase, we choose 2 random positions of the pattern.

In the first phase let the two positions chosen be 2 and 3. In this case, $P' = *\,5\,6\,*$. We run the 1 mismatch algorithm with T and P'. At the end of this phase we realize that $t_3 \neq p_2; t_5 \neq p_2; t_7 \neq p_3; t_{11} \neq p_2; t_{11} \neq p_3; t_{13} \neq p_3; t_{16} \neq p_3$; and $t_{17} \neq p_3$. The corresponding E_i values will be decremented by $e_{i,j}$ values. Specifically, if $t_i \neq p_j$, then E_{i-j+1} is decremented by $e_{i,j}$. For example, since $t_7 \neq p_3$, we decrement E_5 by $e_{7,3} = (6-3)^2 \times 6 \times 3 = 162$. E_5 becomes zero, and hence, T_5 is output as a correct answer. Likewise, since $t_{11} \neq p_3$, we decrement E_9 by $e_{11,3} = (1-6)^2 \times 1 \times 6 = 150$. Now, E_9 becomes 32.

In the second phase, let the two positions chosen be 1 and 2. In this case $P' = 2\,5\,*\,*$. At the end of this phase, we learn that $t_7 \neq p_2; t_9 \neq p_1; t_{11} \neq p_1; t_{13} \neq p_2; t_{15} \neq p_1; t_{16} \neq p_1$. Here, again, relevant E_i values are decremented. For instance, since $t_9 \neq p_1$, E_9 is decremented by $e_{9,1} = (4-2)^2 \times 4 \times 2 = 32$. The value of E_9 now becomes zero, and hence, T_9 is output as a correct answer; and so on.

If the distance between T_i and P (for some i) is $\leq k$, then out of all of the phases attempted, there is a high probability that all of these mismatches between T_i and P will be identified.

The authors of [17] also present an improved algorithm whose run time is $O(n \log m(k + \log n \log\log n))$. The main idea is the observation that if $q_i = k$ for any i, then in $O(k \log n)$ sampling steps, we can identify $\geq k/2$ mismatches. There are several iterations where in each iteration $O(k + \log n)$, sampling phases are done. At the end of each iteration, the value of k is changed to $k/2$. Let this algorithm be called Algorithm 6.

2.4.3. A Las Vegas Algorithm

In this section, we present a Las Vegas algorithm for the k mismatches problem when there are wild cards in the text and/or the pattern. This algorithm runs in time $\widetilde{O}(nk \log^2 m + n \log^2 m \log n + n \log m \log n \log\log n)$. This algorithm is based on the algorithm of [17]. When the algorithm terminates, for each i $(1 \leq i \leq (n - m + 1))$, either we would have identified all of the q_i mismatches between T_i and P or we would have identified more than k mismatches between T_i and P.

Algorithm 5 will be used for every i for which $q_i \leq 2k$. For every i for which $q_i > 2k$, we use the following strategy. Let $2^\ell k < q_i \leq 2^{\ell+1} k$ (where $1 \leq \ell \leq \log\left(\left\lfloor \frac{m}{2k} \right\rfloor\right)$). Let $w = \log\left(\left\lfloor \frac{m}{2k} \right\rfloor\right)$. There will be w phases in the algorithm, and in each phase, we perform $O(k)$ sampling steps. Each sampling step in phase ℓ involves choosing $\frac{m}{2^{\ell+1}k}$ positions of the pattern uniformly at random (for $1 \leq \ell \leq w$). As we show below, if for any i, q_i is in the interval $[2^\ell, 2^{\ell+1}]$, then at least k mismatches between T_i and P will be found in phase ℓ with high probability. The pseudocode for the algorithm is given in Algorithm 7.

Theorem 6. *Algorithm 7 runs in time* $\widetilde{O}(nk \log^2 m + n \log^2 m \log n + n \log m \log n \log\log n)$ *if Algorithm 6 is used in Step 1. It runs in time* $\widetilde{O}(nk \log m \log n + nk \log^2 m + n \log^2 m \log n)$ *if Step 1 uses Algorithm 5.*

Algorithm 7:

while *true* **do**

1. Run Algorithm 5 or Algorithm 6;
2. **for** $\ell \leftarrow 1$ **to** w **do**
   - **for** $r \leftarrow 1$ **to** ck *(c being a constant)* **do**
     - Uniformly randomly choose $\frac{m}{2^{\ell+1}k}$ positions of the pattern;
     - Generate a string P', such that $|P'| = |P|$ and P' has the same characters as P in these randomly chosen positions and zero everywhere else;
     - Run the 1 mismatch algorithm on T and P';
     - As a result, if there is a single mismatch between T_i and P', then add the position of mismatch to $L(i)$ and reduce the value of E_i by the right amount, for $1 \leq i \leq (n-m+1)$;
3. **if either** $E_i = 0$ **or** $|L(i)| > k$ **for every** i, $1 \leq i \leq (n-m+1)$ **then** quit;

Proof. As shown in [17], the run time of Algorithm 5 is $O(nk \log m \log n)$, and that of Algorithm 6 is $O(n \log m(k + \log n \log\log n))$. The analysis will be done with respect to an arbitrary T_i. In particular, we will show that after the specified amount of time, with high probability, we will either know q_i or realize that $q_i > k$. It will then follow that the same statement holds for every T_i (for $1 \leq i \leq (n-m+1)$.

Consider phase ℓ of Step 2 (for an arbitrary $1 \leq \ell \leq w$). Let $2^\ell k < q_i \leq 2^{\ell+1}k$ for some i. Using the fact that $\binom{a}{b} \approx \left(\frac{ae}{b}\right)^b$, the probability of isolating one of the mismatches in one run of the sampling step is:

$$\frac{\binom{m-q_i}{m/(2^{\ell+1}k)-1} q_i}{\binom{m}{m/(2^{\ell+1}k)}} \geq \frac{\binom{m-2^{\ell+1}k}{m/(2^{\ell+1}k)-1} 2^\ell k}{\binom{m}{m/(2^{\ell+1}k)}} \geq \frac{1}{2e}$$

As a result, using Chernoff bounds (Equation (3) with $\delta = 1/2$, for example), it follows that if $13ke$ sampling steps are made in phase ℓ, then at least $6k$ of these steps will result in the isolation of single mismatches (not all of them need be distinct) with high probability (assuming that $k = \Omega(\log n)$). Moreover, we can see that at least $1.1k$ of these mismatches will be distinct. This is because the probability that $\leq 1.1k$ of these are distinct is $\leq \binom{q_i}{1.1k} / \left(\frac{1.1k}{q_i}\right)^{6k} \leq 2^{-2.64k}$ using the fact that $q_i \geq 2k$. This probability will be very low when $k = \Omega(\log n)$.

In the above analysis, we have assumed that $k = \Omega(\log n)$. If this is not the case, in any phase of Step 2, we can do $c\alpha \log n$ sampling steps, for some suitable constant c. In this case, also we can perform an analysis similar to that of the above case using Chernoff bounds. Specifically, we can show that with high probability, we will be able to identify all of the mismatches between T_i and P. As a result, each phase of Step 2 takes $O(n \log m(k + \log n))$ time. We have $O(\log m)$ phases. Thus, the run time of Step 2 is $O(n \log^2 m(k + \log n))$. Furthermore, the probability that the condition in Step 3 holds is very high.

Therefore, the run time of the entire algorithm is $\widetilde{O}(nk \log^2 m + n \log^2 m \log n + n \log m \log n \log\log n)$ if Algorithm 6 is used in Step 1 or $\widetilde{O}(nk \log m \log n + nk \log^2 m + n \log^2 m \log n)$ if Algorithm 5 is used in Step 1. □

3. Results

The above algorithms are based on symbol comparison, arithmetic operations or a combination of both. Therefore, it is interesting to see how these algorithms compare in practice.

In this section, we compare deterministic algorithms for pattern matching. Some of these algorithms solve the pattern matching with mismatches problem, and others solve the k mismatches problem. For the sake of comparison, we treated all of them as algorithms for the k mismatches problem, which is a special case of the pattern matching with mismatches problem.

We implemented the following algorithms: the naive $O(nm)$ time algorithm, Abrahamson's algorithm [10], subset k mismatches (Section 2.2.2) and knapsack k mismatches (Section 2.2.3). For subset k mismatches, we simulate the suffix tree and LCA extensions by a suffix array with an LCP (longest common prefix; [25]) table and data structures to perform RMQ queries (range minimum queries; [26]) on it. This adds a $O(\log n)$ factor to preprocessing. For searching in the suffix array, we use a simple forward traversal with a cost of $O(\log n)$ per character. The traversal uses binary search to find the interval of suffixes that start with the first character of the pattern. Then, another binary search is performed to find the suffixes that start with the first two characters of the pattern, and so on. However, more efficient implementations are possible (e.g., [27]). For subset k mismatches, we also tried a simple $O(m^2)$ time pre-processing using dynamic programming to precompute LCPs and hashing to quickly determine whether a portion of the text is present in the pattern. This method takes more preprocessing time, but it does not have the $O(\log n)$ factor when searching. Knapsack k mismatches uses subset k mismatches as a subroutine, so we have two versions of it, as well.

We tested the algorithms on protein, DNA and English inputs generated randomly. We randomly selected a substring of length m from the text and used it as the pattern. The algorithms were tested on an Intel Core i7 machine with 8 GB of RAM, Linux Mint 17.1 Operating System and gcc 4.8.2. All convolutions were performed using the fftw[28] library Version 3.3.3. We used the suffix array algorithm RadixSAof [29].

Figure 1 shows run times for varying the length of the text n. All algorithms scale linearly with the length of the text. Figure 2 shows run times for varying the length of the pattern m. Abrahamson's algorithm is expensive, because, for alphabet sizes smaller than $\sqrt{m/\log m}$, it computes one convolution for every character in the alphabet. The convolutions proved to be expensive in practice, so Abrahamson's algorithm was competitive only for DNA data, where the alphabet is small. Figure 3 shows runtimes for varying the maximum number of mismatches k allowed. The naive algorithm and Abrahamson's algorithm do not depend on k; therefore, their runtime is constant. Subset k mismatch, with its $O(nk)$ runtime, is competitive for relatively small k. Knapsack k mismatch, on the other hand, scaled very well with k. Figure 4 shows runtimes for varying the alphabet from four (DNA) to 20 (protein) to 26 (English). As expected, Abrahamson's algorithm is the most sensitive to the alphabet size.

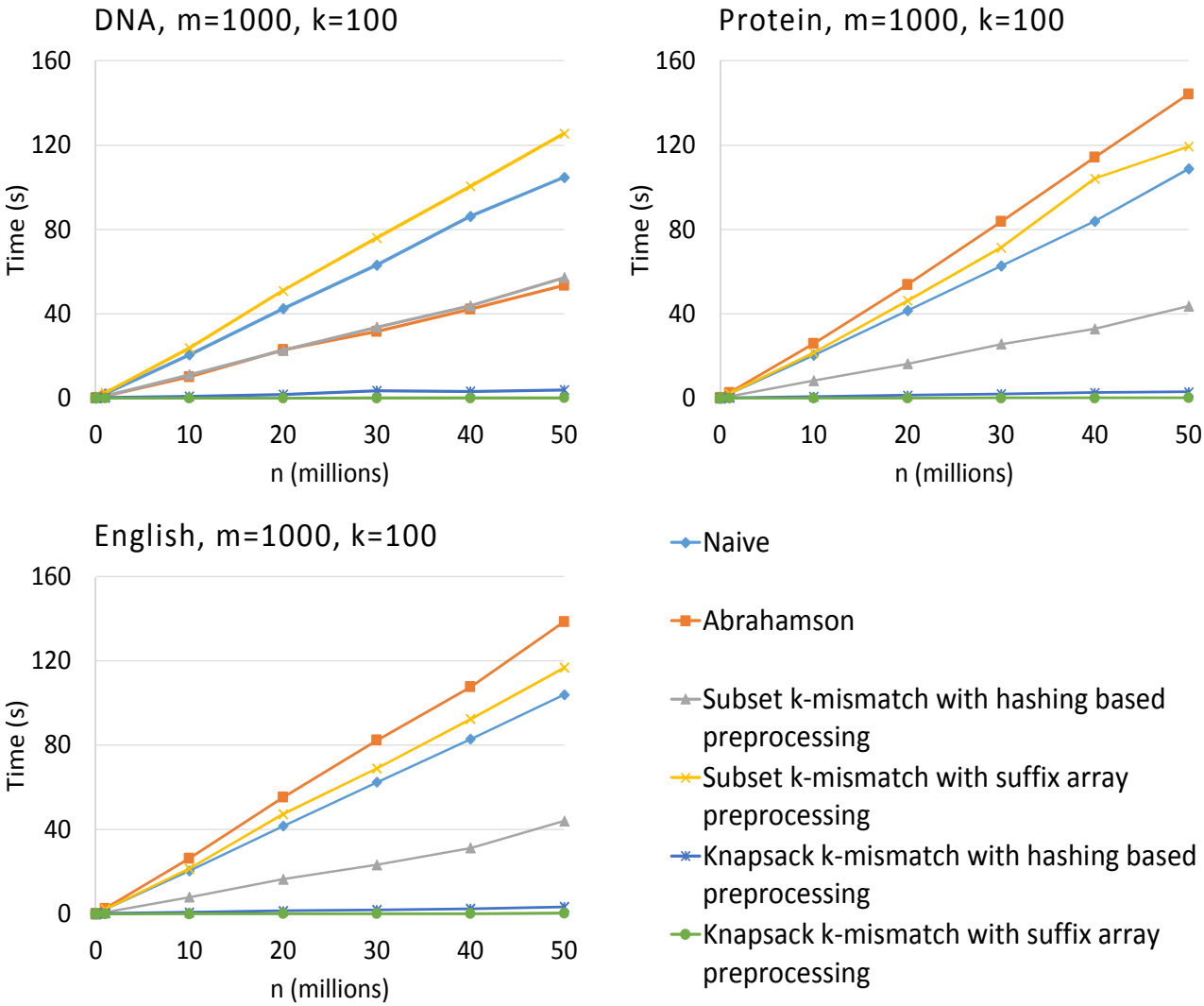

Figure 1. Run times for pattern matching on DNA, protein and English alphabet data, when the length of the text (n) varies. The length of the pattern is $m = 1000$. The maximum number of mismatches allowed is $k = 100$. Our algorithms are subset k mismatch with hashing-based preprocessing (Section 2.2.2), subset k mismatch with suffix array preprocessing (Section 2.2.2), knapsack k mismatch with hashing-based preprocessing (Section 2.2.3) and knapsack k mismatch with suffix array preprocessing (Section 2.2.3).

Overall, the naive algorithm performed well in practice, most likely due to its simplicity and cache locality. Abrahamson's algorithm was competitive only for small alphabet size or for large k. Subset k mismatches performed well for relatively small k. In most cases, the suffix array version was slower than the hashing-based one with $O(m^2)$ time pre-processing because of the added $O(\log n)$ factor when searching in the suffix array. It would be interesting to investigate how the algorithms compare with a more efficient implementation of the suffix array. Knapsack k mismatches was the fastest among the algorithms compared, because in most cases, the knapsack could be filled with less than the given "budget", and thus, the algorithm did not have to perform any convolution operations.

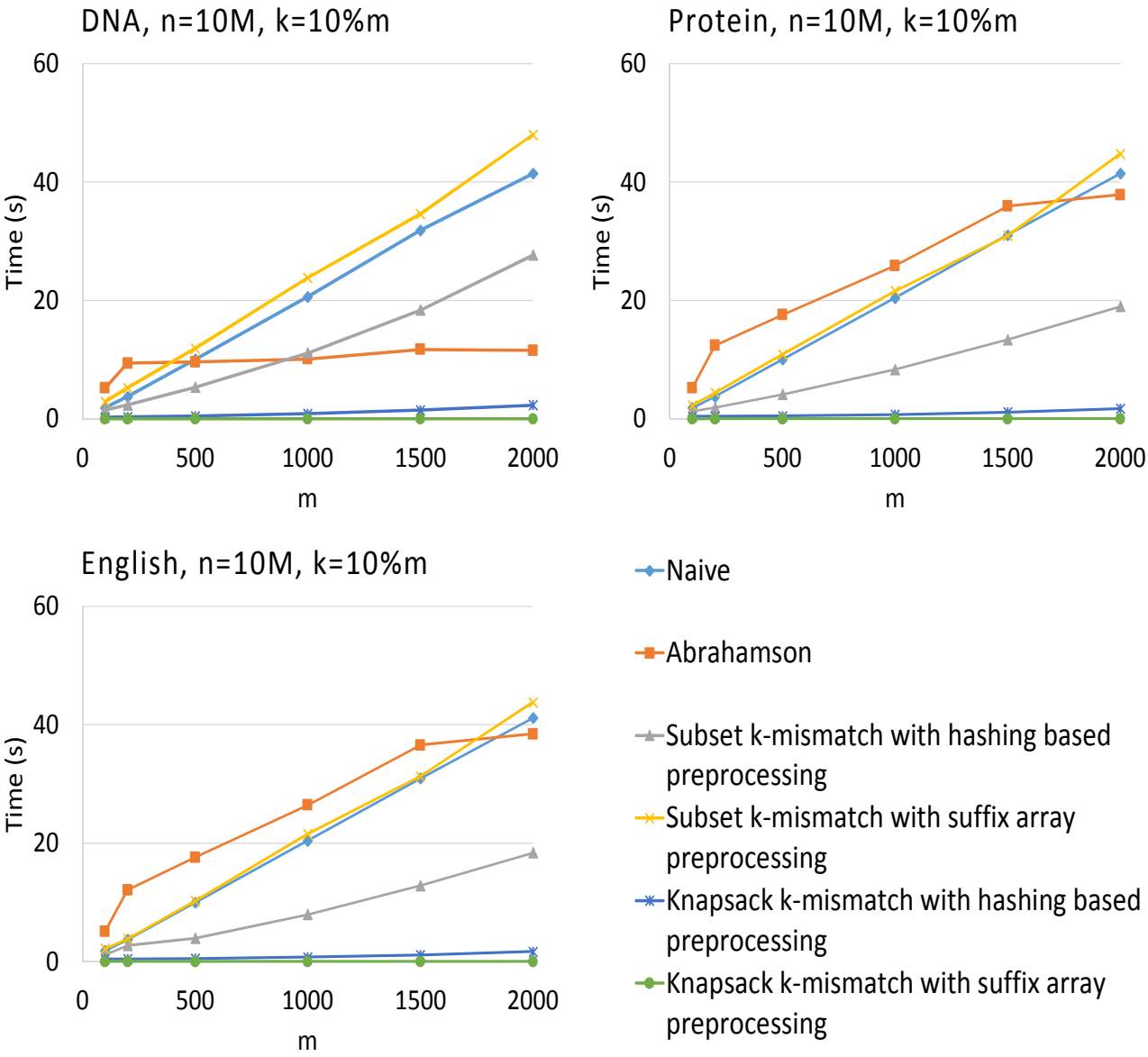

Figure 2. Run times for pattern matching on DNA, protein and English alphabet data, when the length of the pattern (m) varies. The length of the text is $n = 10$ millions. The maximum number of mismatches allowed is $k = 10\%$ of the pattern length. Our algorithms are subset k mismatch with hashing-based preprocessing (Section 2.2.2), subset k mismatch with suffix array preprocessing (Section 2.2.2), knapsack k mismatch with hashing-based preprocessing (Section 2.2.3) and knapsack k mismatch with suffix array preprocessing (Section 2.2.3).

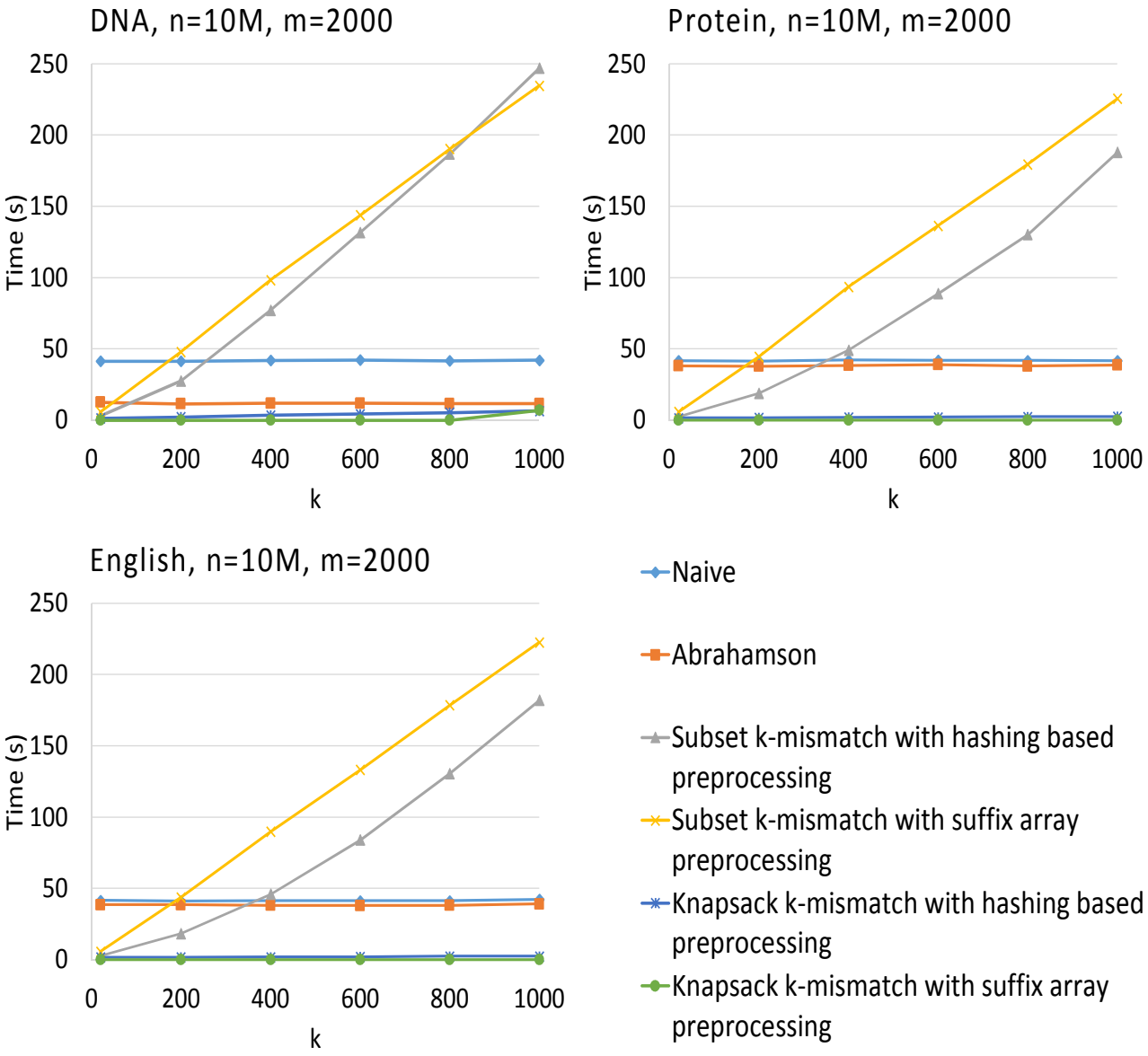

Figure 3. Run times for pattern matching on DNA, protein and English alphabet data, when the maximum number of mismatches allowed (k) varies. The length of the text is $n = 10$ millions. The length of the pattern is $m = 2000$. Our algorithms are subset k mismatch with hashing-based preprocessing (Section 2.2.2), subset k mismatch with suffix array preprocessing (Section 2.2.2), knapsack k mismatch with hashing-based preprocessing (Section 2.2.3) and knapsack k mismatch with suffix array preprocessing (Section 2.2.3).

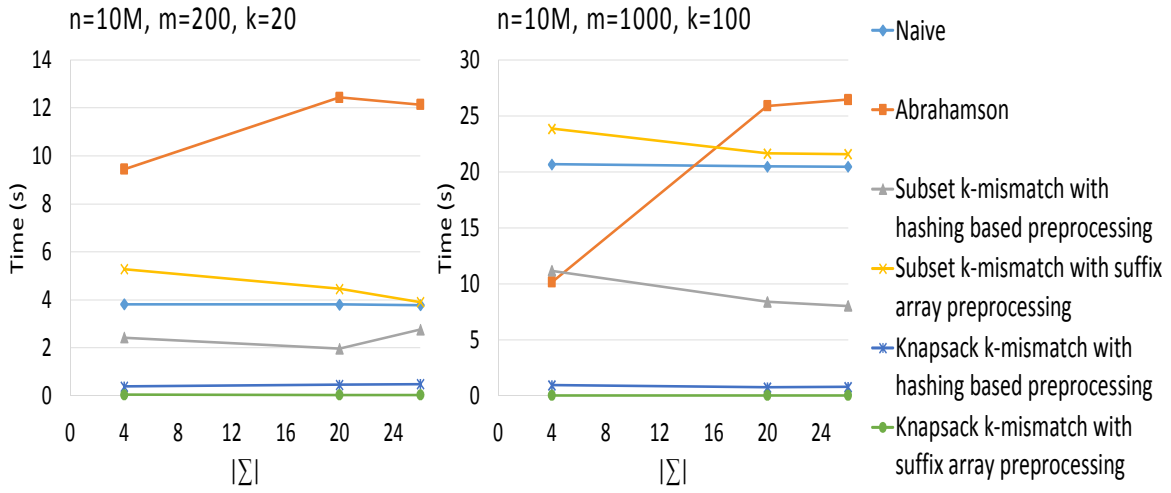

Figure 4. Run times for pattern matching when the size of the alphabet varies from four (DNA) to 20 (protein) to 26 (English). The length of the text is $n = 10$ millions. The length of the pattern is $m = 200$ in the first graph and $m = 1000$ in the second. The maximum number of mismatches allowed is $k = 20$ in the first graph and $k = 100$ in the second. Our algorithms are subset k mismatch with hashing-based preprocessing (Section 2.2.2), subset k mismatch with suffix array preprocessing (Section 2.2.2), knapsack k mismatch with hashing-based preprocessing (Section 2.2.3) and knapsack k mismatch with suffix array preprocessing (Section 2.2.3).

4. Conclusions

We have introduced several deterministic and randomized, exact and approximate algorithms for pattern matching with mismatches and the k mismatches problems, with or without wild cards. These algorithms improve the run time, simplify or extend previous algorithms wild cards. We have also implemented the deterministic algorithms. An empirical comparison of these algorithms showed that the algorithms based on character comparison outperform those based on convolutions.

Acknowledgments

This work has been supported in part by the following grants: NSF 1447711, NSF 0829916 and NIH R01LM010101.

Author Contributions

Sanguthevar Rajasekaran designed the randomized and approximate algorithms. Marius Nicolae designed and implemented the deterministic algorithms and carried out the empirical experiments. Marius Nicolae and Sanguthevar Rajasekaran drafted and approved the manuscript.

Conflicts of Interest

The authors declare no conflict of interest.

References

1. Knuth, D.E.; Morris, J.H., Jr.; Pratt, V.R. Fast Pattern Matching in Strings. *SIAM J. Comput.* **1977**, *6*, 323–350.
2. Aho, A.V.; Corasick, M.J. Efficient string matching: An aid to bibliographic search. *Commun. ACM* **1975**, *18*, 333–340.
3. Fischer, M.J.; Paterson, M.S. String-Matching and Other Products. Technical Report MAC-TM-41, Massachusetts Institute of Technology Cambridge Project MAC, Cambridge, MA, USA, 1974.
4. Indyk, P. Faster Algorithms for String Matching Problems: Matching the Convolution Bound. In Proceedings of the 39th Symposium on Foundations of Computer Science, Palo Alto, CA, USA, 8–11 November 1998; pp. 166–173.
5. Kalai, A. Efficient pattern-matching with don't cares. In Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms; Society for Industrial and Applied Mathematics: Philadelphia, PA, USA, 2002; SODA '02, pp. 655–656.
6. Clifford, P.; Clifford, R. Simple deterministic wildcard matching. *Inf. Process. Lett.* **2007**, *101*, 53–54.
7. Cole, R.; Hariharan, R.; Indyk, P. Tree pattern matching and subset matching in deterministic O(n log3 n)-time. In Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms, Baltimore, MD, USA, 17–19 January 1999; Society for Industrial and Applied Mathematics: Philadelphia, PA, USA, 1999; SODA '99, pp. 245–254.
8. Navarro, G. A guided tour to approximate string matching. *ACM Comput. Surv.* **2001**, *33*, 31–88.
9. Navarro, G.; Raffinot, M. *Flexible Pattern Matching in Strings–Practical on-Line Search Algorithms for Texts and Biological Sequences*; Cambridge University Press: Cambridge, UK, 2002.
10. Abrahamson, K. Generalized String Matching. *SIAM J. Comput.* **1987**, *16*, 1039–1051.
11. Amir, A.; Lewenstein, M.; Porat, E. Faster algorithms for string matching with k mismatches. *J. Algorithms* **2004**, *50*, 257–275.
12. Atallah, M.J.; Chyzak, F.; Dumas, P. A randomized algorithm for approximate string matching. *Algorithmica* **2001**, *29*, 468–486.
13. Karloff, H. Fast algorithms for approximately counting mismatches. *Inf. Process. Lett.* **1993**, *48*, 53–60.
14. Clifford, R.; Efremenko, K.; Porat, B.; Porat, E. A black box for online approximate pattern matching. In *Combinatorial Pattern Matching*; Springer: Berlin Heidelberg, Germany, 2008; pp. 143–151.
15. Porat, B.; Porat, E. Exact and Approximate Pattern Matching in the Streaming Model. In Proceedings of the (FOCS '09) 50th Annual IEEE Symposium on Foundations of Computer Science, 2009; pp. 315–323.
16. Porat, E.; Lipsky, O. Improved sketching of Hamming distance with error correcting. *Combinatorial Pattern Matching*; Springer: Berlin Heidelberg, Germany, 2007; pp. 173–182.
17. Clifford, R.; Efremenko, K.; Porat, E.; Rothschild, A. k-mismatch with don't cares. *Algorithms–ESA 2007*; Springer: Berlin Heidelberg, Germany, 2007; pp. 151–162.

18. Clifford, R.; Efremenko, K.; Porat, B.; Porat, E.; Rothschild, A. Mismatch Sampling. *Inf. Comput.* **2012**, *214*, 112–118.
19. Landau, G.M.; Vishkin, U. Efficient string matching in the presence of errors. In Proceedings of the 26th Annual Symposium on Foundations of Computer Science, Portland, OR, USA, 21–23 October, 1985; pp. 126 –136.
20. Galil, Z.; Giancarlo, R. Improved string matching with k mismatches. *SIGACT News* **1986**, *17*, 52–54.
21. Clifford, R.; Efremenko, K.; Porat, E.; Rothschild, A. From coding theory to efficient pattern matching. In Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, New York, New York, USA, 4–6 January, 2009; Society for Industrial and Applied Mathematics: Philadelphia, PA, USA, 2009; SODA '09, pp. 778–784.
22. Clifford, R.; Porat, E. A filtering algorithm for k-mismatch with don't cares. *Inf. Process. Lett.* **2010**, *110*, 1021–1025.
23. Fredriksson, K.; Grabowski, S. *Combinatorial Algorithms*; Springer-Verlag: Berlin, Germany, 2009; Chapter Fast Convolutions and Their Applications in Approximate String Matching, pp. 254–265.
24. Chernoff, H. A Measure of Asymptotic Efficiency for Tests of a Hypothesis Based on the Sum of Observations. *Ann. Math. Stat.* **1952**, *2*, 241–256.
25. Kasai, T.; Lee, G.; Arimura, H.; Arikawa, S.; Park, K. *Linear-Time Longest-Common-Prefix Computation in Suffix Arrays And Its Applications*; Springer-Verlag: Berlin, Germany, 2001; pp. 181–192.
26. Bender, M.; Farach-Colton, M. The LCA Problem Revisited. In *LATIN 2000: Theoretical Informatics*; Gonnet, G., Viola, A., Eds.; Springer: Berlin, Germany, 2000; Volume 1776, pp. 88–94.
27. Ferragina, P.; Manzini, G. Opportunistic data structures with applications. In Proceedings of the IEEE 41st Annual Symposium on Foundations of Computer Science, Redondo Beach, CA, USA, 12–14 November, 2000; pp. 390–398.
28. Frigo, M.; Johnson, S.G. The Design and Implementation of FFTW3. *Proc. IEEE* **2005**, *93*, 216–231.
29. Rajasekaran, S.; Nicolae, M. An elegant algorithm for the construction of suffix arrays. *J. Discrete Algorithms* **2014**, *27*, 21–28.