Algorithmica (2000) OF1–OF19
DOI: 10.1007/s004530010062

Algorithmica
© 2000 Springer-Verlag New York Inc.

# A Randomized Algorithm for Approximate String Matching[1]

M. J. Atallah,[2] F. Chyzak,[3] and P. Dumas[3]

**Abstract.** We give a randomized algorithm in deterministic time $O(N \log M)$ for estimating the score vector of matches between a text string of length $N$ and a pattern string of length $M$, i.e., the vector obtained when the pattern is slid along the text, and the number of matches is counted for each position. A direct application is approximate string matching. The randomized algorithm uses convolution to find an estimator of the scores; the variance of the estimator is particularly small for scores that are close to $M$, i.e., for approximate occurrences of the pattern in the text. No assumption is made about the probabilistic characteristics of the input, or about the size of the alphabet. The solution extends to string matching with classes, class complements, "never match" and "always match" symbols, to the weighted case and to higher dimensions.

**Key Words.** Convolution, FFT, Approximate string matching, Randomized algorithms.

## 1. Scores and Approximate String Matching

*Problem Statement*. For a text string $T = t_0 t_1 \cdots t_{N-1}$ and a pattern string $P = p_0 p_1 \cdots p_{M-1}$, we address the problem of computing the *score vector of matches between $T$ and $P$*. This is defined as the vector $C$ whose $i$th component $c_i$ is the number of matches between the text and the pattern when the first letter of the pattern is positioned in front of the $i$th letter of the string (see Figure 1).

A related problem is *approximate string matching*, which consists in finding occurrences of small variations of the pattern string $P$ in the text string $T$. The strings found differ from the pattern by a few insertions, deletions, or substitutions of letters. Computing the score vector solves a version of the problem of approximate string matching where only substitutions are permitted: an exact match corresponds to a score $c = M$; a match with $e$ errors to a score $c = M - e$. In this work we also consider *pattern matching with classes*, where a position in the pattern is allowed to match any letter from a finite class, but do not address the case of searching for *regular expressions*, which would correspond to matching any pattern from a finite class, or any repetition of a given pattern.

[1] The first author gratefully acknowledges support from the CERIAS Center at Purdue and its sponsors. The second and third authors' work was supported in part by the Long Term Research Project Alcom-IT (#20244) of the European Union.
[2] CERIAS and Department of Computer Sciences, Purdue University, West Lafayette, IN 47907, USA. mja@cs.purdue.edu.
[3] INRIA, Rocquencourt, B.P. 105, 78153 Le Chesnay Cedex, France. {Frederic.Chyzak, Philippe.Dumas}@inria.fr.

Received July 20, 1997; revised April 20, 1998, and June 1, 1999. Communicated by Ming-Yang Kao.
Online publication October 5, 2000.

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Position | | | | | | $i$ | | | | | | | | |
| Text | ... | b | c | a | a | b | c | a | a | b | b | b | a | c | ... |
| Pattern | | | | | | a | b | a | b | b | a | | | | |
| Matches | | | | | | | | ↑ | | ↑ | | | | | |

**Fig. 1.** The pattern is slid along the text and for each position we count the number of matches between the pattern and the corresponding slice of the text; this gives the score $C$ (here $C_i = 2$).

Approximate string matching has many applications, including intrusion detection in a computer system [11], image analysis, and data compression [2]. In the first application, alphabet symbols correspond to events in a system, and since some events are more important than others (from a security point of view), the scores require to be *weighted* by the relative importance of alphabet symbols. This leads us to consider a weighted version of the problem which computes *weighted scores*:

$$c_i = \sum_{j=0}^{M-1} w(p_j)\delta_{t_{i+j},p_j}, \qquad 0 \le i \le N - M,$$

where $N \ge M$, $\delta_{x,y}$ denotes the Kronecker symbol, and $w$ is a complex-valued function defined over the alphabet. The basic nonweighted case corresponds to a constant function $w(p) = 1$.

*Method.* Rather than focusing on computing the exact scores, we develop in this paper a *randomized* algorithm of Monte-Carlo type to compute an unbiased estimate of the score vector. The algorithm computes the score vector by a convolution, which makes it possible to use the fast Fourier transform. Although randomized, its behavior neither depends on any a priori probabilistic assumption on the input, nor on the size of the alphabet. It proceeds by computing and averaging $k$ independent equally distributed estimates for the score vector. The expected value of the averaged estimator is equal to the exact value. In other words, the expected value of the $i$th component $\widehat{c_i}$ of our estimate $\widehat{C}$ of the score vector equals $c_i$. It turns out that the standard deviation is bounded by $(M - c_i)/\sqrt{k}$, and that the algorithm can be tuned to attain an arbitrary level of accuracy. Moreover, the fewer the number $M - c_i$ of mismatches, the better the approximation that the algorithm returns: even if the estimated score can differ somewhat from the exact value when the pattern and the text have little match, an almost complete match will be recognized by the algorithm. The latter thus locates interesting positions with good accuracy, and the algorithm can at least theoretically be used as a filter: after a few positions have been recognized as good candidates for approximate matches, the exact scores can be computed for those few positions only. Our method generalizes to the weighted case as well.

*Complexity.* As already mentioned, we obtain an asymptotically fast algorithm by using fast Fourier transform to compute convolutions [9]. As a result our algorithm runs in deterministic time $O(kN\gamma(M)/M)$, where $\gamma(M)$ is the time needed to perform the convolution of two vectors of length $M$. Henceforth, we replace $\gamma(M)$ by $M \log M$ which corresponds to the computational model where an arithmetic operation takes constant time. We thus get an algorithm in deterministic time $O(kN \log M)$. Note the tradeoff between time complexity and accuracy: by choosing larger values of $k$, more accurate estimates are obtained. However, preliminary experiments suggest that small values for $k$ are sufficient in practice to achieve a reasonable accuracy. The following theorem summarizes our main result in the nonweighted case.

THEOREM 1. *An estimate for the score $C$ between a text string of length $N$ and a pattern string of length $M$ can be computed by a Monte-Carlo algorithm in time $O(kN \log M)$, where $k$ is the number of iterations in the algorithm. The randomized result has mean $C$ and each entry has a variance bounded by $(M - c_i)^2/k$.*

*Algorithmical Context.* A continuous and intensive research effort since the 1970s has led to a great deal of approximate string matching algorithms. These algorithms typically have a time complexity linear in the size $N$ of the text, but with a dependency in the size $M$ of the pattern between linear and logarithmical. We proceed to list and sketch the main existing algorithms. All the complexity evaluations below refer to arithmetic complexity and are based on a computational model in which the convolution of two vectors of length $M$ is performed in time $O(M \log M)$. The list splits into three types of algorithms: algorithms based on fast multiplication for large integers, practical algorithms based on hardware, and a more recent generation of randomized algorithms.

The first algorithm that comes to mind for computing exact score vectors is the naive (deterministic) algorithm with a time complexity of $O((N - M + 1)M)$. Several algorithms escape this quadratic complexity by the use of efficient multiplication algorithms for large integers:

- Fischer and Paterson use convolution to solve the special case of finding all exact occurrences (i.e., scores that equal exactly $M$) in the presence of "always match" symbols [7]. This algorithm has time complexity $O(N \log M \log \sigma)$ and requires the alphabet to be fixed, finite, and known beforehand. Although the size of the alphabet used for a text of length $N$ can be as large as $N$, splitting the text in chunks of length $O(M)$ to be dealt with independently ensures it will work with an alphabet size $\sigma = O(M)$, which extends the previous algorithms to the case when alphabets are not known beforehand.
- Abrahamson and Kosaraju independently extended the algorithm by Fischer and Paterson into a deterministic algorithm for computing the vector $C$ in time complexity $O(N\sqrt{M} \log M)$ [1], [10], allowing for generalized alphabets with classes and alphabets that were not known beforehand (see Section 4). Their clever approach makes judicious use of two different methods: convolution to compute the contribution of alphabet symbols that occur frequently; a more direct (and quite straightforward) method to compute the contribution of alphabet symbols that occur infrequently. Note that Abrahamson also gives a variant algorithm of time complexity $O(N \log M)$ restricted to fixed finite alphabets known beforehand and allowing classes [1].

Of a different nature, several asymptotically slow algorithms make crucial use of the hardware in order to lessen their practical complexity: in spite of a bad theoretical complexity in $O(MN)$ they beat the other algorithms in practice for small sizes. This is the case of the next two algorithms which more generally deal with approximate string matching when insertion and deletion are also allowed:

- The algorithm of Baeza-Yates and Gonnet solves the problem in time $O(NM \log M/\log N)$ [4], which is better than $O(N \log M)$ for very small $M$, i.e., $M = o(\log N)$. Besides, for even smaller values of $M$, say $M = O(1)$, this algorithm has a very low practical complexity, linear in $N$ with a low constant factor, because all parameters of the algorithm can then be packed on the same machine word and be processed using very few hardware operations.
- The algorithm of Baeza-Yates and Perleberg solves the problem in time $O(NMf_{\max})$ where $f_{\max}$ is the maximal occurrence frequency of symbols in the pattern [5]. The idea of the algorithm is to rely on fast operations on linked lists. For patterns ruled by an equiprobable Bernoulli model, the average time complexity is $O(NM/\sigma)$, which is good for large alphabets when the pattern size $M$ is fixed. However, in view of a fair comparison with other algorithms when $M$ is large, note that maintaining that the complexity of this algorithm is small, say $N \times o(M)$, requires $f_{\max}$ to tend to zero, and $\sigma$ to grow unbounded with $M$.

The interest in the vector $C$ is usually motivated in applications by the need to find all positions in the text at which the pattern *almost occurs*, i.e., the offsets $i$ such that $c_i$ is close to $M$. From this viewpoint, computing exact values for the scores is not needed. A recent trend is the introduction of randomization in the computation of scores.

- An algorithm of deterministic time $O(N \log M)$ was given in [3], whose analysis depends on some restrictive assumptions on the probabilistic characteristics of the input, namely the Bernoulli model. Although this model is not realistic, the contribution of this paper is the introduction of randomization in the problem of approximate string matching, together with a hashing of the alphabet which allows one to reduce to working with a fixed alphabet that is known beforehand.
- As opposed to Fisher and Paterson, Karloff studied the case when the alphabet is not known beforehand and gave a clever deterministic algorithm of time $O(N \log^3 M)$ for estimating all the scores of mismatches [8]. He also provided a randomized variant of deterministic time complexity $O(N \log^2 M)$. Karloff's estimator is *intentionally biased* in order to guarantee not overestimating the number of mismatches by more than a constant multiplicative factor. The method apparently cannot be modified to estimate the number of matches (rather than of mismatches).

## 2. Description of the Algorithm.

Assume that we have two strings of length $M$ over a finite alphabet $\Sigma$ of cardinality $\sigma$. The algorithm is based on the following idea: if we renumber the letters by the application of a map $\Phi$ from the alphabet to the integer interval

$$[0, \sigma) = \{0, \dots, \sigma - 1\},$$

we obtain two integer sequences $n_0 \cdots n_{M-1}$ and $m_0 \cdots m_{M-1}$. Now note that a match between the two strings at position $j$ induces a match $n_j = m_j$. This contributes 1 in the

Hermitian inner product

$$\sum_{j=0}^{M-1} \omega^{n_j} \overline{\omega^{m_j}} = \sum_{j=0}^{M-1} \omega^{n_j - m_j},$$

where $\omega$ denotes any primitive $\sigma$th root of unity. On the other hand a mismatch contributes a perturbative term $\omega^{n_j - m_j}$.

From the nullity of the sum of all the $\sigma$th roots of unity, one observes that the mean $\mathrm{E}(\omega^X)$ is zero when $X$ is a uniformly distributed random variable over $[0, \sigma)$. Consequentially we introduce the set $\Xi$ of all possible mappings from $\Sigma$ to $[0, \sigma)$, and turn $\Phi$ into a uniformly distributed random variable over $\Xi$ so as to obtain the score between both strings as the mean of the Hermitian inner product over all renumberings.

As to our problem of computing the score vector $C$, we could make use of the previous idea to compute each of its $N - M + 1$ entries successively. However, this would ineluctably lead to a time complexity of $O(NM)$. The turning point of our method is to interpret the score vector as the mean over all letter renumberings of the *convolution* of two randomized finite sequences of complex numbers. In this way, the *simultaneous* calculation of all the $c_i$'s is made possible by the use of fast Fourier transform (FFT). Additionally, we apply the standard technique [6] of partitioning the text into overlapping chunks of size $(1+\alpha)M$ each, and then processing each chunk separately. Processing one chunk supplies $\alpha M$ components of $C$, so that we need no more than $N/(\alpha M)$ chunks. In this discussion the parameter $\alpha$ may depend on $M$. We choose it to be $O(M)$ and larger than a constant, so that each chunk requires a time $O((1+\alpha)M \log((1+\alpha)M))$. The time complexity for one iteration step is therefore

$$\begin{aligned} &\frac{N}{\alpha M} O((1+\alpha)M \log((1+\alpha)M)) \\ &\qquad = O\left(\frac{1+\alpha}{\alpha} N \log((1+\alpha)M)\right) = O(N \log M). \end{aligned}$$

The overall time complexity of our Monte-Carlo algorithm is then $O(kM \log M)$ where $k$ is the number of repetitions performed. The basic case $N = (1+\alpha)M$ of the algorithm is sketched in Figure 2. We name it MC after the Monte-Carlo approach used.

The end of the section is devoted to further comments and to variations of the algorithm.

A closer look at the dependency in $\alpha$ of the complexity permits us to minimize the implied constant in the big oh. To this end, introduce the constant $\tau$ defined by the complexity $\gamma(M) = \tau M \log M$ of the FFT. For the simple choice $\alpha = 1$, the complexity reduces to $2\tau N \log(2M)$. The better choice $\alpha = \log M$ lessens the constant 2 to $1 + (\log\log M)/\log M$, up to terms of order $O(1/\log M)$. This suggests an optimal choice of $\alpha = \Theta(\log M)$.

Roots of unity appear in two different ways in the algorithm. On the one hand $\sigma$th roots of unity $\omega^i$ are used to encode the alphabet into complex numbers. On the other hand the FFT to compute the convolution of two complex vectors of size $(1+\alpha)M$ uses roots of unity $\zeta^i$ of order a power of two which is not smaller than $(1+\alpha)M$. It should be clear that $\omega$ is not related to $\zeta$, for the alphabet size and the text size are independent from one another. In spite of the fixed precision of the numerical computations, first

```
Input: a text $T = t_0 \cdots t_{(1+\alpha)M-1}$ and a pattern $P = p_0 \cdots p_{M-1}$ where the
  $t_i$'s and the $p_i$'s are letters from $\Sigma$;
Output: an estimate for the score vector $C$.

  1. For $\ell = 1, 2, \ldots, k$:
     (a) randomly and uniformly select a $\Phi^{(\ell)}$ from $\Xi = [0, \sigma)^{\Sigma}$;
     (b) from the text $T$, obtain a complex sequence $T^{(\ell)}$ of size $(1+\alpha)M$
         by replacing every symbol $t$ in $T$ by $\omega^{\Phi^{(\ell)}(t)}$;
     (c) from the pattern $P$, obtain a complex sequence $P^{(\ell)}$ by
          i. replacing every symbol $p$ in $P$ by $\omega^{-\Phi^{(\ell)}(p)}$;
         ii. padding with $\alpha M$ (trailing) zeros;
     (d) compute the vector $C^{(\ell)}$ as the convolution of $T^{(\ell)}$ with the reverse
         of $P^{(\ell)}$, i.e.,
         $$c_i^{(\ell)} = \sum_{j=0}^{M-1} \omega^{\Phi^{(\ell)}(t_{i+j})} \overline{\omega^{\Phi^{(\ell)}(p_j)}} = \sum_{j=0}^{M-1} \omega^{\Phi^{(\ell)}(t_{i+j}) - \Phi^{(\ell)}(p_j)};$$
  2. compute the vector $\widehat{C} = \sum_{\ell=1}^{k} C^{(\ell)}/k$ and output it as an estimate of $C$.
```

**Fig. 2.** Algorithm MC.

experimental results (Section 5) show that the round-off error causes no apparent loss of validity of the theoretical predictions. Yet, an unavoidable restriction is that the precision of the numerical calculations be less than the inverse of the product $2(1+\alpha)kM\sigma$.

Note that one could encode the alphabet into a finite field instead of the complex, and use FFT in this framework. This would avoid any round-off errors, but would require fields $\mathbb{F}_{p^2}$ for a large prime $p$, and to compute with the same number of bits as in the complex case.

**3. Probabilistic Analysis of the Output Estimate.** We now study the mean and the variance of the estimators $\widehat{c_i}$. It turns out that the mean $\mathrm{E}(\widehat{c_i})$ is $c_i$, and that the standard deviation of $\widehat{c_i}$ is bounded by $(M - c_i)/\sqrt{k}$. This result was already summarized in Theorem 1.

All the random variables $\widehat{c_i}$ are defined in a similar way; hence we generically consider the random variable

$$\hat{s} = \frac{1}{k} \sum_{\ell=1}^{k} \sum_{j=0}^{M-1} \omega^{\Phi^{(\ell)}(t_j) - \Phi^{(\ell)}(p_j)},$$

where the $t_j$'s and the $p_j$'s are fixed and the mappings $\Phi^{(\ell)}$'s are independent and uniformly distributed random mappings from $\Sigma$ to $[0, \sigma)$. The number $c$ of matches between $t_0 \cdots t_{M-1}$ and $p_0 \cdots p_{M-1}$ is

$$c = \sum_{j=0}^{M-1} \delta_{t_j, p_j}.$$

The random variable $\hat{s}$ is the mean of $k$ independent identically distributed random variables $s^{(\ell)}$. Hence it suffices to consider the random variable

$$s = \sum_{j=0}^{M-1} \omega^{\Phi(t_j)-\Phi(p_j)},$$

for the mean and variance of $\hat{s}$ are then

$$\mathrm{E}\big(\hat{s}\big) = \mathrm{E}(s) \quad \text{and} \quad \mathrm{Var}\big(\hat{s}\big) = \frac{\mathrm{Var}(s)}{k}.$$

We start by evaluating the mean of $\hat{s}$ with the following lemma.

LEMMA 1. *The mean of $\hat{s}$ is the number $c$ of matches between $t_0 \cdots t_{M-1}$ and $p_0 \cdots p_{M-1}$.*

PROOF. The mean of $\hat{s}$ is

$$\mathrm{E}(\hat{s}) = \mathrm{E}(s) = \sum_{j=0}^{M-1} \mathrm{E}(\omega^{\Phi(t_j)-\Phi(p_j)}).$$

Now, observe that the mean inside the sum is zero unless $t_j = p_j$, because $\omega^{\Phi(t_j)-\Phi(p_j)}$ is equally likely to be any of the $\sigma$th roots of unity, whose sum is zero. More precisely, we have the equality

$$\mathrm{E}(\omega^{\Phi(t_j)-\Phi(p_j)}) = \delta_{t_j,p_j},$$

from which the result follows. □

Next, we consider the variance of $\hat{s}$. We mention the corresponding result now for the purpose of exposition, but postpone its proof to the next section where it is proved in more generality.

LEMMA 2. *The variance of $\hat{s}$ is bounded as follows*:

$$\mathrm{Var}\big(\hat{s}\big) \leq \frac{(M-c)^2}{k}.$$

Theorem 1 now follows from Lemmata 1 and 2.

## 4. Generalized String Matching.

We extend the previous technique in several directions. The main contribution here is to show that classical generalizations also apply to our algorithm and to perform the corresponding complexity analyses. The first extension to be analyzed is a weighted version of the problem. This allows for more general functions than the characteristic function of matches, and is used by the other extensions. Then we show how our algorithm extends to pattern matching of arrays in place of words, or more generally to higher-dimensional arrays. Next, a different extension of our algorithm allows us to accommodate classes of letters, class complements, "never match" and "always match" symbols in the patterns and when possible in the texts. For the simplicity of the exposition, we present each extension separately, but they could all be merged in the same algorithm and implementation.

*Weighted Case.* The method and results we developed apply to weighted versions of the problem, i.e., to the problem of computing weighted scores defined by

$$c_i = \sum_{j=0}^{M-1} w(p_j)\delta_{t_{i+j},p_j},$$

where $w$ is a complex-valued function defined over the alphabet. In fact, we consider scores of the form

$$c_i = \sum_{j=0}^{M-1} f(t_{i+j})g(p_j)\delta_{t_{i+j},p_j},$$

where $f$ and $g$ are complex-valued functions defined over the alphabet. These two formulations may seem equivalent. Nonetheless, we use the second formulation because it suggests a better intuition of the algorithm and enables the further extensions of the next sections.

In the algorithm, the encoding of the alphabet using roots of unity has to be changed accordingly: when creating $T^{(\ell)}$ we now replace every symbol $t$ in $T$ by $f(t)\omega^{\Phi^{(\ell)}(t)}$, while when creating $P^{(\ell)}$ we replace every symbol $p$ in $P$ by $g(p)\omega^{-\Phi^{(\ell)}(p)}$.

As a matter of fact, we proceed to perform our analysis in the even more general case of weighted scores of the form

$$c_i = \sum_{j=0}^{M-1} h(t_{i+j}, p_j)\delta_{t_{i+j},p_j},$$

where $h$ is a complex-valued function on pairs of letters in $\Sigma^2$. We do this essentially for the purpose of mathematical analysis, although our convolution-based algorithm can only deal with the special case of $h(a,b) = f(a)g(b)$. The randomized vector $\widehat{C}$ we obtain still has the property to be $C$ in the mean, and the variance of $\widehat{c_i}$ to be $O((M-c_i)/\sqrt{k})$. However, the restriction to the special case of weights is crucially needed from the computational point of view to represent, and compute, the vector score by a convolution.

Once again, we generically consider the random variable

$$\hat{s} = \frac{1}{k}\sum_{\ell=1}^{k}\sum_{j=0}^{M-1} h(t_j, p_j)\omega^{\Phi^{(\ell)}(t_j)-\Phi^{(\ell)}(p_j)},$$

where the $t_j$'s and the $p_j$'s are fixed and the $\Phi^{(\ell)}$'s are independent and uniformly distributed random mappings from $\Sigma$ to $[0,\sigma)$. The weighted score between $t_0\cdots t_{M-1}$ and $p_0\cdots p_{M-1}$ is

$$c = \sum_{j=0}^{M-1} h(t_j, p_j)\delta_{t_j,p_j}.$$

The random variable $\hat{s}$ is the mean of $k$ independent identically distributed random variables $s^{(\ell)}$. Hence it suffices to consider the random variable

$$s = \sum_{j=0}^{M-1} h(t_j, p_j)\omega^{\Phi(t_j)-\Phi(p_j)},$$

for a single random renumbering $\Phi$. The mean and variance of $\hat{s}$ are then

$$\mathrm{E}(\hat{s}) = \mathrm{E}(s) \quad \text{and} \quad \mathrm{Var}\big(\hat{s}\big) = \frac{\mathrm{Var}(s)}{k}.$$

The analysis differs from the unweighted case in that the role of $\delta_{x,y}$ in the unweighted case is now played by $h(x, y)\delta_{x,y}$. We start with the mean.

LEMMA 3. *The mean of $\hat{s}$ is the weighted score*

$$c = \sum_{j=0}^{M-1} h(t_j, p_j)\delta_{t_j,p_j}$$

*between $t_0 \cdots t_{M-1}$ and $p_0 \cdots p_{M-1}$.*

PROOF. The mean of $\hat{s}$ is

$$\begin{aligned}\mathrm{E}(\hat{s}) = \mathrm{E}(s) &= \sum_{j=0}^{M-1} \mathrm{E}(h(t_j, p_j)\omega^{\Phi(t_j)-\Phi(p_j)}) \\ &= \sum_{j=0}^{M-1} h(t_j, p_j)\mathrm{E}(\omega^{\Phi(t_j)-\Phi(p_j)}) = c,\end{aligned}$$

since $\mathrm{E}(\omega^{\Phi(t_j)-\Phi(p_j)}) = \delta_{t_j,p_j}$. □

We now turn to the variance, proving Lemma 2 as a particular case.

LEMMA 4. *The variance of $\hat{s}$ is bounded by*

$$\mathrm{Var}\left(\hat{s}\right) \leq \frac{||h||_\infty (M-c)^2}{k},$$

*where $||h||_\infty$ denotes the maximum value of $|h(x, y)|$ over $\Sigma^2$.*

PROOF. To express $\mathrm{Var}(s) = \mathrm{E}(|s|^2) - |\mathrm{E}(s)|^2$, we first derive an explicit form for the mean of $|s|^2 = s\bar{s}$, starting with the equality

$$\mathrm{E}(s\bar{s}) = \sum_{0\leq i,j<M} h(t_i, p_i)\overline{h(t_j, p_j)}\mathrm{E}(\omega^{\Phi(t_i)-\Phi(p_i)-\Phi(t_j)+\Phi(p_j)}).$$

When $\omega^{\Phi(t_i)-\Phi(p_i)-\Phi(t_j)+\Phi(p_j)} = 1$ independently from $\Phi$, i.e., when $t_i = t_j$ and $p_i = p_j$, or when $t_i = p_i$ and $t_j = p_j$, the inner mean

$$\mathrm{E}(\omega^{\Phi(t_i)-\Phi(p_i)-\Phi(t_j)+\Phi(p_j)})$$

is 1; otherwise, it is 0.

By a simple inclusion–exclusion argument, it follows that

$$\mathrm{E}(s\bar{s}) = \sum_{0\leq i,j<M} h(t_i, p_i)\overline{h(t_j, p_j)}(\delta_{t_i,p_i}\delta_{t_j,p_j} + \delta_{t_i,t_j}\delta_{p_i,p_j} - \delta_{t_i,t_j}\delta_{p_i,p_j}\delta_{t_i,p_i}\delta_{t_j,p_j}).$$

With the first product of Kronecker symbols, one recognizes the expansion of $|\mathrm{E}(s)|^2$, so that

$$\begin{aligned}\mathrm{Var}(s) &= \mathrm{E}(|s|^2) - |\mathrm{E}(s)|^2\\ &= \sum_{0\le i,j<M} h(t_i, p_i)\overline{h(t_j, p_j)}\delta_{t_i,t_j}\delta_{p_i,p_j}(1-\delta_{t_i,p_i}\delta_{t_j,p_j}).\end{aligned}$$

We introduce the real symmetric matrix $\Gamma = [\gamma_{i,j}]$ of size $\sigma \times \sigma$ with $(i, j)$th entry given by

$$\gamma_{i,j} = \delta_{t_i,t_j}\delta_{p_i,p_j}(1-\delta_{t_i,p_i}\delta_{t_j,p_j}),$$

and the vector $H$ with $i$th entry $h(t_i, p_i)$. We obtain $\mathrm{Var}(s) = \overline{H}^{\mathrm{T}}\Gamma H$, where T denotes the transpose of matrices. Call $\rho(\Gamma)$ the spectral radius of $\Gamma$, i.e., the largest modulus of its eigenvalues. Since $\Gamma$ is positive semidefinite (because it describes variances), its eigenvalues are nonnegative and $\rho(\Gamma)$ is the largest eigenvalue. We have

$$\mathrm{Var}(s) = \overline{H}^{\mathrm{T}}\Gamma H \le \rho(\Gamma)\overline{H}^{\mathrm{T}}H.$$

To improve on the previous upper bound and make it more explicit, we need to take the number $c$ of matches into account.

The number $\gamma_{i,j}$ is 0 unless $t_i = t_j \ne p_i = p_j$. It entails that in case of a match $t_i = p_i$, both the $i$th row and the $i$th column of $\Gamma$ are 0. After renumbering the rows and columns in $\Gamma$ and $H$, we part them as follows:

$$\Gamma = \left[\begin{array}{c|c} 0 & 0 \\ \hline 0 & \Gamma' \end{array}\right] \quad \text{and} \quad H = \left[\begin{array}{c} * \\ \hline H' \end{array}\right],$$

where $\Gamma' = [\gamma'_{i,j}]$ is a matrix of size $(M-c)\times(M-c)$ and $H'$ is a vector of size $(M-c)$. It follows that

$$\overline{H}^{\mathrm{T}}\Gamma H = \overline{H'}^{\mathrm{T}}\Gamma' H' \le \rho(\Gamma')\overline{H'}^{\mathrm{T}}H'.$$

On the other hand, the spectral radius $\rho(\Gamma')$ of $\Gamma'$ is bounded by the Schur norm $\mathcal{N}(\Gamma')$ which is defined by

$$\mathcal{N}(\Gamma')^2 = \sum_{1\le i,j\le M-c} |\gamma'_{i,j}|^2 \le (M-c)^2.$$

Furthermore, setting

$$||h||_\infty = \max_{(x,y)\in\Sigma^2} |h(x, y)|,$$

we obtain

$$\overline{H'}^{\mathrm{T}}H' \le ||h||_\infty^2(M-c).$$

Finally,

$$\mathrm{Var}(\hat{s}) = \frac{\mathrm{Var}(s)}{k} = \frac{\overline{H}^{\mathrm{T}}\Gamma H}{k} = \frac{\overline{H'}^{\mathrm{T}}\Gamma' H'}{k} \le \frac{||h||_\infty^2(M-c)^2}{k}. \qquad \square$$

The two lemmata above together prove the following theorem.

THEOREM 2. *For the weighted version of the problem, an estimate $\widehat{C}$ for the score $C$ between a text string of length $N$ and a pattern string of length $M$ can be computed by a Monte-Carlo algorithm in deterministic time $O(kN \log M)$ with mean and variance*

$$\mathrm{E}(\widehat{C}) = C \quad \textit{and} \quad \mathrm{Var}(\widehat{c_i}) \leq \frac{||h||_\infty^2 (M - c_i)^2}{k}.$$

Most commonly when the weights are defined by a single function $w$ as in the introduction of this section,

$$||h||_\infty = ||w||_\infty = \max_{x \in \Sigma} |w(x)|.$$

Also note that the variance is once again particularly small when $c_i$ is close to $M$.

*Higher-Dimensional Arrays*. We sketch the extension to two-dimensional arrays in the nonweighted case; similar ideas would extend it to three and higher dimensions, and to mixed weighted higher-dimensional versions as well.

For the sake of simplicity, we assume in what follows that $M$ and $N$ are the squares of two integers, $M = m^2$ and $N = n^2$, and that $N \geq M$. The text $T$ is now a matrix of size $n \times n$, the pattern $P$ is a smaller matrix of size $m \times m$, and the result we seek is an $(n + 1 - m) \times (n + 1 - m)$ matrix $C$ where

$$c_{i,j} = \sum_{k=0}^{m-1} \sum_{l=0}^{m-1} \delta_{T_{i+k,j+l}, P_{k,l}},$$

for $0 \leq i, j \leq n - m$. The time to compute our estimate $\widehat{C}$ of $C$ is now $O(kN \log M)$, and we still have $\mathrm{E}(\widehat{C}) = C$ and $\mathrm{Var}(\widehat{c_{i,j}}) \leq (M - c_{i,j})^2/k$. We next briefly sketch how this is done.

We justify our focus to achieving a time complexity of $O(kM \log M)$ for the case $n = 2m$ by the following standard reduction [6] to this case from the general case $n > 2m$:

- Cover $T$ with $N/M$ overlapping squares $T_{i,j}$ of size $2m \times 2m$ each, where $T_{i,j}$ consists of the square submatrix of $T$ of size $2m \times 2m$ that has its top-left corner at position $(im, jm)$ in $T$. Hence $T_{i,j}$ and $T_{i+1,j+1}$ overlap over a region of $T$ of size $m \times m$, $T_{i,j}$ and $T_{i,j+1}$ overlap over a region of size $2m \times m$, and $T_{i,j}$ and $T_{i+1,j}$ overlap over a region of size $m \times 2m$.
- The algorithm for the case $n = 2m$ is then used on each of the $N/M$ pairs $(T_{i,j}, P)$ of text and pattern. It is easy to see that these $N/M$ answers together contain a description of the desired matrix $C$. The overall time complexity to compute them is $O((N/M)kM \log M) = O(kN \log M)$, as required.

Therefore, we henceforth assume that $n = 2m$.

The extension of the one-dimensional solution to two dimensions works by transforming the two-dimensional problem into a one-dimensional one [6], and in the process introduces "never match" symbols: that is, if $\Sigma$ is the alphabet for the two-dimensional problem, then the corresponding alphabet for the one-dimensional problem is $\Sigma \cup \{\#\}$ where #

is a “never match” symbol in the sense that if $x$ or $y$ (or both) equal #, then $\delta_{x,y} = 0$ as a convention.

More specifically, from the text matrix $T$ of size $2m \times 2m$, we create the corresponding text vector $V$ by concatenating the rows of $T$. Thus $V$ has length $4m^2$. From the pattern matrix $P$ of size $m \times m$, we create a pattern vector $W$ of length $2m^2$ by augmenting each of the rows of $P$ by appending to the end of each of them $m$ symbols # and then concatenating the augmented rows. Let $K$ be the score vector with $V$ as text and $W$ as pattern, i.e.,

$$K_i = \sum_{j=0}^{2m^2-1} \delta_{V_{i+j}, W_j}$$

for $0 \le i \le 2m^2$ and with the understanding that $\delta_{x,y}$ is zero if either $x$ or $y$ equals the special symbol #.

The connection between $K$ and the score matrix $C$ for text $T$ and pattern $P$ is now given by

$$c_{i,j} = K_{2m(i-1)+j}.$$

Therefore, computing the matrix $C$ reduces to computing the vector $K$. The computation is not complicated much more by the presence of the new, special # symbol: we simply follow the rules of Algorithm MC except that, at the place where the algorithm is required to create $\omega^{\Phi(t)}$ (resp. $\omega^{-\Phi(p)}$), we only do so if $t$ (resp. $p$) is not the # symbol, and we create a 0 instead if $t$ (resp. $p$) is the # symbol. Hence we use the weighted model introduced in the previous section, with the weight functions

$$f(a) = g(a) = 1$$

for any letter $a \in \Sigma$ except from

$$f(\#) = g(\#) = 0.$$

Lemmata 1 and 2 simply lead to the following theorem.

THEOREM 3. *For the two-dimensional version of the problem, an estimate $\widehat{C}$ for the score array $C$ between a text array of size $n \times n$ (for $n^2 = N$) and a pattern array of size $m \times m$ (for $m^2 = M$) can be computed by a Monte-Carlo algorithm in deterministic time $O(kN \log M)$ with mean and variance*

$$\mathrm{E}(\widehat{C}) = C \quad \textit{and} \quad \mathrm{Var}(\widehat{c_{i,j}}) \le \frac{(M - c_{i,j})^2}{k}.$$

*String Matching with Classes*. Let $a_v$ be letters in an alphabet. By a *class* $[a_1 \cdots a_r]$ we mean a new symbol that matches any of the letters $a_v$. In particular, letters can be viewed as classes: the classes consisting in a single letter. Another special class is the *full class*, i.e., the class consisting of the whole alphabet.

We first restrict to allowing classes in either the text or the pattern. Without loss of generality, we focus on classes in patterns. Each symbol $p_j$ in a pattern $P = p_0 \cdots p_{M-1}$

is now a class $[p_{j,1} \cdots p_{j,r_j}]$. We modify our algorithm by replacing each class $p_j$ in the pattern by

$$\sum_{v=1}^{r_j} w(p_{j,v})\omega^{-\Phi^{(\ell)}(p_{j,v})}$$

while creating $P^{(\ell)}$. The convolution vector $C^{(\ell)}$ is thus

$$c_i^{(\ell)} = \sum_{j=0}^{M-1} \sum_{v=1}^{r_j} w(p_{j,v})\omega^{\Phi^{(\ell)}(t_{i+j})}\omega^{-\Phi^{(\ell)}(p_{j,v})},$$

so that the modified algorithm still has the same time and space complexities. For mean and variance analyses, we once again generically consider the random variable

$$s = \sum_{j=0}^{M-1} \sum_{v=1}^{r_j} w(p_{j,v})\omega^{\Phi(t_j)}\omega^{-\Phi(p_{j,v})}.$$

By the linearity of the mean, we have

$$\mathrm{E}(s) = \sum_{j=0}^{M-1} \sum_{v=1}^{r_j} w(p_{j,v})\delta_{t_j, p_{j,v}}.$$

For the variance analysis, we mentally replicate $r_j$ times each $t_j$ in the text, while mentally replacing each $p_j$ by $p_{j,1} \cdots p_{j,r_j}$ in the pattern. We are thus led to two strings of length $M' = \sum_j r_j$, whole convolution yields the same score as above. This yields the following theorem.

THEOREM 4. *When allowing classes in the pattern, an estimate $\widehat{C}$ for the score $C$ between a text string of length $N$ and a pattern string $p = p_0 \cdots p_{M-1}$ of length $M$ for classes $p_j = [p_{j,0} \cdots p_{j,r_j}]$ can be computed by a Monte-Carlo algorithm in deterministic time $O(kN \log M)$ with mean and variance*

$$\mathrm{E}(\widehat{C}) = C \quad \textit{and} \quad \mathrm{Var}(\widehat{c_i}) \le \frac{||h||_\infty^2 (M' - c_i)^2}{k} \qquad \textit{for} \quad M' = \sum_{j=1}^{M} r_j \ge M.$$

So far, we have only weighted letters uniformly with respect to positions in the text and in the pattern, by the weight function $w$. It is additionally possible to weight letters within a class, allowing different weights for the same letter according to its position in the text or in the pattern: we denote by

$$\left[\sum_{i=1}^{r} p_i a_i\right]$$

the *weighted class* consisting of the letters $a_i$'s weighted by the $p_i$'s. This notion extends that of classes, since we have

$$\left[\sum_{i=1}^{r} a_i\right] = [a_1 \cdots a_r].$$

As another example, the $p_i$'s can be viewed as probabilities when the $p_i$'s add up to 1. It is then possible to allow classes both in the pattern and in the text, and to get a consistent interpretation for this: for a second weighted class

$$\left[\sum_{i=1}^{s} q_i b_i\right],$$

we define the match between both classes to be

$$\sum_{i=1}^{r}\sum_{j=1}^{s} p_i q_j \delta_{a_i,b_j}.$$

In this probabilistic interpretation, the score counts the matches according to the probability of occurrence of each letter in each class. Algorithmically, computing this score by Algorithm MC is achieved by using $f$ to encode the $p_i$'s and $g$ to encode the $q_i$'s.

*"Never Match" and "Always Match" Symbols.* To allow more flexible string matching on a given alphabet $\Sigma$ and achieve a better accuracy of the estimates, we adjoin two new special symbols, a "never match" symbol # and an "always match" symbol $\star$.

The "never match" symbol # was already introduced in the previous section. It corresponds to a symbol that nevers matches any other letter; in other words, it satisfies $\delta_{a,\#} = \delta_{\#,a} = 0$ for any letter $a \in \Sigma$. It may be used simultaneously in the pattern and in the text and the weighted model extends to this new symbol by simply assuming

$$f(\#) = g(\#) = 0.$$

Working with the extended alphabet $\Sigma \cup \{\#\}$ does not change the analysis of the previous sections.

The "always match" symbol $\star$ corresponds to a symbol that matches any other letter; in other words, it satisfies $\delta_{a,\star} = \delta_{\star,a} = 1$ for any letter $a \in \Sigma$. It may be used simultaneously in the pattern and in the text and the weighted model extends to this new symbol by simply assuming

$$f(\star) = 1 \quad \text{and} \quad g(\star) = w(\star).$$

In this respect, it is very much like the full class (the class consisting of all the elements of the alphabet). Still, it is of a different nature, differing in the way weights are dealt with. Only as an exception, both notions share the same semantics in the simple case when no weights are used, i.e., when matches are counted by ones and mismatches by zeros ($w = 1$). As a convention, the "always match" symbol matches itself; whether "always match" and "never match" symbols match each other is irrelevant in what follows. In all the cases, we get an algorithm that is only four times slower to yield sharper estimates. This algorithm (Figure 3) is based on four applications of our algorithm in the following way. Let $u$ be a new symbol, which we adjoin to the alphabet and view as a letter (i.e., it only matches itself).

As an optimization, steps 2 and 3 of the algorithm could be avoided when "always match" symbols are not used in the text (and when "never match" symbols do not match

*Input*: a text $T = t_0 \cdots t_{2M-1}$ and a pattern $P = p_0 \cdots p_{M-1}$ where the $t_i$'s and the $p_i$'s are letters from $\Sigma \cup \{*\}$;
*Output*: an estimate for the score vector $C$.

1. Replace all * by # in both the text and the pattern, and apply Algorithm MC; this matches letters in the text against letters in the pattern;
2. replace all * by $u$ and all letters by # in the text, and all letters by $u$ and all * by # in the pattern, and apply Algorithm MC; this matches "always match" symbols in the text against letters in the pattern;
3. replace all letters by $u$ and all * by # in the text, and all * by $u$ and all letters by # in the pattern, and apply Algorithm MC; this matches letters in the text against "always match" symbols in the pattern;
4. replace all * by $u$ in both the text and the pattern and all letters by #, and apply Algorithm MC; this matches "always match" symbols in the text against "always match" symbols in the pattern;
5. add the four estimates previously obtained and return the sum as the result of the algorithm.

**Fig. 3.** Algorithm MC with "always match" symbols.

"always match" symbols). In this case, one would simply count the number of "always match" symbols in the pattern, and add the corresponding contribution to the result. The algorithm then only requires twice as much time as the original Algorithm MC.

Noting that steps 2–4 yield exact estimates makes the analysis of the algorithm easy, and we obtain the following theorem.

THEOREM 5. *When allowing "never match" and "always match" both in the text and in the pattern, an estimate $\widehat{C}$ for the score $C$ between a text string of length $N$ and a pattern string of length $M$ can be computed by a Monte-Carlo algorithm in deterministic time $O(kN \log M)$ with mean and variance*

$$\mathrm{E}(\widehat{C}) = C \quad \textit{and} \quad \mathrm{Var}(\widehat{c_i}) \le \frac{||h||_\infty^2 (M - c_i)^2}{k}.$$

To compare the analyses obtained when using the "always match" symbol * or the full class, consider the extreme case of a pattern consisting of $M$ symbols * (and disallow # in the text). The variance obtained by Theorem 5 is then zero, since $c_i = M$ for all $i$. Now, consider a pattern made of $M$ copies of the full class. The variance obtained by Theorem 4 is now

$$\frac{||h||_\infty^2 (\sigma - 1)^2 M^2}{k},$$

where $\sigma$ is the cardinality of the alphabet. Consequently, the use of the full class introduces a lot of noise for a not too small alphabet, due to the randomization of the algorithm. To achieve reasonable variances anyway then requires a number $k$ of iterations of the algorithm of the order of $\sigma^2$, thus to increase the time complexity. The same phenomenon arises in fact for all "large" classes, i.e., classes with cardinality close to $\sigma$; this motivates the next section.

*Input*: a text $T = t_0 \cdots t_{2M-1}$ and a pattern $P = p_0 \cdots p_{M-1}$ where the $t_i$'s and the $p_i$'s are letters from $\Sigma \cup \{*\}$, classes or class complements over $\Sigma$;
*Output*: an estimate for the score vector $C$.

1. Replace all * by # and each class complement $\overline{[a_1 \cdots a_r]}$ by the weighted class $[\sum_{i=1}^{r} - a_i]$ in both the text and the pattern, and apply Algorithm MC;
2. replace all * and all class complement by $u$ and all letters by # in the text, and all letters by $u$ and all * and all class complement by # in the pattern, and apply Algorithm MC;
3. replace all letters by $u$ and all * and all class complement by # in the text, and all * and all class complement by $u$ and all letters by # in the pattern, and apply Algorithm MC;
4. replace all * and all class complement by $u$ in both the text and the pattern and all letters by #, and apply Algorithm MC;
5. add the four estimates previously obtained and return the sum as the result of the algorithm.

**Fig. 4.** Algorithm MC with class complements.

*Class Complements*. Allowing "large" classes in the pattern or in the text yields large variances, as described by the formulae in Theorem 4. To avoid this, it can be advantageous in some cases to describe each "large" class as the *complement* of a "small" class: for a class $[a_1 \cdots a_r]$, we introduce a new symbol $\overline{[a_1 \cdots a_r]}$ that matches any letters but the $a_v$'s. As we previously did for classes, we allow class complements in the patterns only, in order to get a sensical interpretation. Moreover, we deal with a nonweighted alphabet. In this case, the match of a letter $b$ against the class complement $\overline{[a_1 \cdots a_r]}$ is counted by the match of $b$ against * minus the match of $b$ against $[a_1 \cdots a_r]$. This yields the algorithm shown in Figure 4, where class complements are basically viewed as an "always match" symbol and a correcting contribution is removed at step 1.

Theorem 4 still holds after replacing "classes" by "classes and class complements."

## 5. Implementation and Experimentation.

We have implemented and tested our algorithm, performing several experiments on several types of data: randomly generated text, sequenced genes, domains in proteins, literature in several natural languages, and MIDI encoding of classical music. Although the kind of string matching without insertion and deletion discussed here may not always be well suited in real-life applications, the examples chosen verify the robustness of the algorithm on a broad variety of sample types. What is observed is excellent agreement of the experiments with the phenomena predicted by the theory: the algorithm returns accurate results as soon as the pattern is sufficiently large (typically, of size $M$ larger than 32 or 64 bytes), even for a small value of the parameter $k$ that controls the number of repetitions in the algorithm. Typically, a number of $k = 3$ repetitions suffices for small values of $M$, and $k = 1$ already yields reliable results for patterns of size $M = 256$ or more if we disregard the scores that correspond to more than 20% of the estimated mismatch.

*Randomly Generated Text*. In order to demonstrate the accuracy of the approximations of our algorithm, a first experiment was performed with randomly generated text and pattern according to a Bernoulli model. Specifically, a text of 8192 bytes was drawn at random according to the uniform distribution over the ASCII alphabet. The first 4096 bytes were picked, and a pattern was obtained by modifying them at random, so as to keep 4042 matches. For this case, the parameters are $N = 2M = 8192$, $\sigma = 256$, and we performed the algorithm for $k = 1$, 2, and 3. We obtained and compared the estimated scores with the corresponding exact scores. Apart from the almost complete match with exact score 4042, all other positions have an exact score less than or equal to 59. The almost complete match is detected by an estimated score with an error of less than 0.2%. Additionally the program behaves well on all other shifts, with scores that were not overestimated by more than a factor of 5: all other estimated scores are not more than $5 \times 60 = 300$, which is much less than 4096.

*Classical Music*. The next experiment shows that that the accuracy of the algorithm is preserved in the case of small patterns in a real-life application: we considered the search for approximate occurrences of a clarinet theme in Beethoven's Fifth Symphony. The experiment consisted in selecting a theme from the symphony and searching for slices in the whole musical piece that are similar to this theme. The data were MIDI code, and the length of the selected theme was $M = 128$, so that we set $N = 2M = 256$, $\sigma = 128$, and we ran the algorithm for $k = 1$, 2, and 3. A threshold of $\lambda = 0.5$ was used in order to filter out approximate matches (i.e., the program outputs only those matches with $c \geq \lambda M$). Selected parts of the output for $k = 3$, sorted by decreasing scores, are displayed in Figure 5. Each block corresponds to a slice of the text which approximately matches the theme. For each block, the field `estd` gives the estimated score $\widehat{c_i}/M$, the field `exact` gives the exact score $c_i/M$, and the field `ratio` gives the ratio $\widehat{c_i}/c_i$. Note the accuracy of the algorithm on this execution: the ratio $\widehat{c_i}/c_i$ varies little around 1. Each approximate match is illustrated by a string which locates the matches between musical notes: the symbols `*` and `-` represent a match and a mismatch, respectively. Beside the exact occurrence of the theme (first block), we catch several occurrences where the pattern and the text almost match during long sequences (next four blocks), as well as an occurrence where intermediate-sized sequences of exact match are interlaced with sequences of full mismatch (last block).

```
estd = 1.000000; exact= 128/128 = 1.000000; ratio=1.000000
********************************************************************************************************************************

estd = 0.753018; exact= 88/128 = 0.687500; ratio=1.095299
-****-***-****-***-****-****-****-***-****-***-****-***-****-****--***--**--***--**--***--**--***--***--***--***--***--***-****-

estd = 0.550580; exact= 70/128 = 0.546875; ratio=1.006775
------------------*----*-*--*-****-***-****-***-****-***-****-*--*-****-***--***-***--***-***--***-*-**--***-****--***-*--**---*

estd = 0.507331; exact= 69/128 = 0.539062; ratio=0.941137
*-****-***-****-***-*--*-*--*-****-***-****-***-****-***-*--*-*--*--***-***--***-***--***-***--***-*--*----*-*------------------

estd = 0.568381; exact= 65/128 = 0.507812; ratio=1.119273
-****-***-****-***-****-****-****-***-****-***-****-***-****-**----***--**---**--**---**--**----*--*---------*------------------

estd = 0.526136; exact= 61/128 = 0.476562; ratio=1.104024
*******************-*--*-*-----------*******************-*--*-*-----------*--***-***--***-****---***----------------------------
```

**Fig. 5.** Searching for a clarinet theme in Beethoven's Fifth Symphony.

*Sequenced Proteins*. Algorithm MC has a low asymptotical complexity but is not meant to be a fast practical one. In a last experiment, we proposed evaluating thresholds with respect to the size $M$ of the patterns for which it becomes faster than the other ones. To this end, we performed the search of prefixes of a sequenced protein in a protein database using implementations for the naive algorithm, the "shift-add" algorithm by Baeza-Yates and Gonnet [4], the "counting" algorithm by Baeza-Yates and Perleberg [5], and our Monte-Carlo algorithm. Although the proteins were encoded over an alphabet with a few dozen symbols only, the database also contained various information and comments, so that we kept the alphabet consisting of all ASCII codes (with $\sigma = 128$). Beside this, we allowed "never match" symbols both in the patterns and in the text, since the protein database we used contained such "never match" symbols. The patterns used were of size $M = 2^p$ for $p$ from 7 to 12 (from $M = 256$ to $M = 4096$).

For each of the algorithms, the observed time complexity closely agrees with the theory. From numerical regression formulae we obtain that our Monte-Carlo algorithm becomes faster than the "shift-add" approach for patterns larger than a few hundreds, and faster than the naive algorithm beyond roughly twice as large a threshold. The fact that the "shift-add" method is roughly twice as slow as the naive algorithm comes from the large values of $M$ that we used, and does not contradict the observation in [4] that the "shift-add" method gets three times faster than the naive algorithm for $M < 9$. Indeed, this phenomenon stems from the fact that several parameters of the algorithm can only be packed in the same machine word and processed simultaneously for very small $M$.

As far as the "counting" algorithm is concerned, the threshold heavily depends on the parameter $f_{\max}$. In the experiment, the value observed for $f_{\max}$ was $\frac{1}{2}$; indeed, the symbols in the patterns appear with frequency less than 5% except for one special padding symbol of frequency $\frac{1}{2}$. The threshold obtained in this situation is roughly 2 kB. A simple extrapolation yields the thresholds 12, 150, and 410 kB for values $f_{\max} = \frac{1}{10}$, $\frac{1}{100}$, and $\frac{1}{256}$, respectively. As already mentioned, we used a soft implementation of FFT which suffers from large constant factors in its time complexity. Although our algorithm should work better with the FFT step performed by dedicated chips, even a speed-up of a factor 1000 in our implementation would not yield a threshold lower than 3 kB for $f_{\max} = \frac{1}{100}$.

**Acknowledgment.** The authors warmly thank Roberto Sierra who sequenced the whole symphony in MIDI code, together with other musical pieces, and made them freely available on the WEB.

## References

[1] Abrahamson, K. Generalized string matching. *SIAM J. Comput.* **16**(6) (1987), 1039–1051.

[2] Atallah, M. J., Génin, Y., and Szpankowski, W. A pattern matching approach to image compression. In *Proceedings of the Third IEEE International Conference on Image Processing*, Lausanne, 1996, pp. 349–356.

[3] Atallah, M. J., Jacquet, P., and Szpankowski, W. Pattern matching with mismatches: a simple randomized algorithm and its analysis. In *Combinatorial Pattern Matching* (*Proceedings of the Third Annual Symposium held in Tucson*, *Arizona*, *April* 29–*May* 1, 1992), A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, eds., vol. 644 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1992, pp. 27–40.

[4] Baeza-Yates, R. A., and Gonnet, G. H. A new approach to text searching. *Comm. ACM* **35** (1992), 74–82.

[5] Baeza-Yates, R. A., and Perleberg, C. H. Fast and practical approximate string matching. *Inform. Process. Lett.* **59**(1) (1996), 21–27.

[6] Crochemore, M., and Rytter, W. *Text Algorithms*. The Clarendon Press, New York, 1994. With a preface by Z. Galil.

[7] Fischer, M. J., and Paterson, M. S. String-matching and other products. In *Complexity of Computation* (*Proceedings of the SIAM–AMS Applied Mathematics Symposium, New York*, 1973), SIAM–AMS Proceedings, Vol. VII, American Mathematical Society, Providence, RI, 1974, pp. 113–125.

[8] Karloff, H. Fast algorithms for approximately counting mismatches. *Inform. Process. Lett.* **48**(2) (1993), 53–60.

[9] Knuth, D. E. *The Art of Computer Programming*, Vol. 2, second edn., Series in Computer Science and Information Processing, Addison-Wesley, Reading, MA, 1981.

[10] Kosaraju, S. R. Efficient string matching. Manuscript, Johns Hopkins University, 1987.

[11] Kumar, S., and Spafford, E. H. A pattern-matching model for intrusion detection. In *Proceedings of the National Computer Security Conference*, 1994, pp. 11–21.