1996

# An Algorithm for Estimating All Matches Between Two Strings

Mikhail J. Atallah
*Purdue University*, mja@cs.purdue.edu

Frédéric Chyzak

Philippe Dumas

Report Number:
96-059

# AN ALGORITHM FOR ESTIMATING ALL MATCHES BETWEEN TWO STRINGS

Mikhail J. Atallah
Frederic Chyzak
Phillippe Dumas

CSD-TR #96-059
October 1996
(Revised April 1998)

# An Algorithm for Estimating all Matches Between Two Strings

Mikhail J. Atallah*
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907
U.S.A.
mja@cs.purdue.edu

Frédéric Chyzak†
INRIA
Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Frederic.Chyzak@inria.fr

Philippe Dumas
INRIA
Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Philippe.Dumas@inria.fr

**Abstract**

We give a randomized algorithm for estimating the score vector of matches between a text string of length $N$ and a pattern string of length $M$; this is the vector obtained when the pattern is slid along the text, and the number of matches is counted for each position. The randomized algorithm takes deterministic time $O((N/M)\mathrm{Conv}(M))$ where $\mathrm{Conv}(M)$ is the time for performing a convolution of two vectors of size $M$ each. In particular, using the fast Fourier transform for convolutions and thus assuming that all arithmetic operations take constant time yields a time complexity of $O(N \log M)$. The algorithm finds an unbiased estimator of the scores, whose variance is particularly small for scores that are close to $M$, i.e., for approximate occurrences of the pattern in the text. No assumptions are made about the probabilistic characteristics of the input, or about the number of different symbols appearing in the text and in the pattern (i.e., the alphabet size need not be much smaller than $M$). The solution extends to string matching with classes, class complements, "never match" and "always match" symbols, to the weighted case and to higher dimensions. We also perform an experimental comparison to a naive string matching algorithm and to a classical algorithm by Baeza-Yates and Gonnet, with the conclusion that our algorithm is faster for patterns of typical length a few thousands or more.

Index Terms — algorithms, convolution, pattern matching.

*This author gratefully acknowledges support from the COAST Project 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.

# 1 Introduction

We address the following problem: let $T$ be a text string and $P$ be a pattern string defined by

$$T = t_0 t_1 \dots t_{N-1} \qquad \text{and} \qquad P = p_0 p_1 \dots p_{M-1} \qquad (N \geq M).$$

We want to compute the *score of matches between $T$ and $P$*, i.e., 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). Formally, for $i = 0, \cdots, N - M$,

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

where $\delta_{x,y}$ denotes the Kronecker symbol: $\delta_{x,y}$ is 1 if and only if $x = y$ and is 0 otherwise.

Computing the score vector solves a version of the problem of approximate pattern matching: an exact match corresponds to a score $c = M$; a match with $e$ errors to a score $c = M - e$. Rather than focusing on computing the exact scores, we develop an efficient randomized algorithm to approximate them. This algorithm can be tuned to attain an arbitrary level of accuracy. Moreover, the fewer the number $e$ of mismatches, the better the approximation that the algorithm returns: even if the estimated score can be far from the exact value when the pattern and the text have little match, an almost complete match will be recognized by the algorithm. The algorithm locates these interesting positions with good accuracy, which is the difficult part of the problem. It can thus 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.

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

Figure 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 $C$.

Approximate pattern matching has many applications, including intrusion detection in a computer system [11], image analysis and data compression [3]. In the former, alphabet symbols correspond to events in a system, and since some events are more important than others (from a security point of view) it follows that the definition of the score needs to be *weighted* by the relative importance of alphabet symbols. This led us to generalize our method and result so that they 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.

The naive (deterministic) algorithm to compute the exact score vector has a time complexity of $O((N - M + 1)M)$. When the alphabet size is $O(1)$ (hence much smaller than $M$), the algorithm of Fischer and Paterson [8] uses convolution to solve the problem in $O(N \log M)$ time. However, if the assumption of small alphabet size is dropped, then another approach is needed. This version of

the problem (i.e., for possibly large alphabets) was posed by Apostolico and Galil in their book [1], where it is mentioned that a linear time algorithm can be obtained for computing those offsets $i$ at which only a single mismatch prevents the pattern from occurring exactly. (The corresponding entries of $C$ then equal $M-1$.) The best known deterministic algorithm for computing the vector $C$ is due independently to Abrahamson and Kosaraju [2, 10] and has a time complexity of $O\left(N\sqrt{M\log M}\right)$ in the arithmetic computational model in which the convolution of two vectors of length $M$ can be done in time $O(M\log M)$. The algorithm of Baeza-Yates and Gonnet [5] solves the problem in $O(NM\log M/\log N)$ time, which is better than $O(N\log M)$ for very small $M$, i.e., if $M=o(\log N)$. 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 [6] solves the problem in average time $O(NM/\sigma)$ where $\sigma$ is the size of the alphabet (i.e., the number of distinct symbols that appear in $M$), which is good for large $\sigma$. The interest in the vector $C$ is usually motivated 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 $M$. An algorithm of probabilistic time $O(N\log M)$ for this problem was given in [4]; however, this algorithm depends on some restrictive assumptions on the probabilistic characteristics of the input, namely the Bernoulli model. (An earlier version of [4] erroneously claimed that such an assumption is not needed by that algorithm, whereas in fact it was needed.) A clever $O(N\log^3 M)$ deterministic time algorithm for estimating all the scores of mismatches (rather than of matches) was given by Karloff [9]; although Karloff's estimator is biased, it guarantees not to overestimate the number of mismatches by more than a constant multiplicative factor, and the author states that his scheme apparently cannot be modified to estimate the number of matches (rather than mismatches) to within a constant multiplicative bound.

In this paper we give a *randomized* algorithm for computing an *unbiased* estimator of the number of matches:

- Our algorithm runs in deterministic time $O((N/M)\mathrm{Conv}(M))$, where $\mathrm{Conv}(M)$ is the time needed to perform the convolution of two vectors of length $M$ all of whose entries are of the form $\omega^j$, where $j$ is an integer and $\omega$ is any primitive $\sigma$th root of unity (recall that $\sigma$ is the number of distinct symbols that appear in $P$). Note that this $\omega$ is not related to the roots of unity used in the Fourier transform implementation of convolution (the order of these roots of unity depends on $M$ rather than on $\sigma$, which is typically smaller than $M$). In the rest of the paper, we replace $\mathrm{Conv}(M)$ by $M\log M$, which corresponds to the computational model where an arithmetic operation (addition or multiplication) takes constant time. Our experimental implementations use $\omega=e^{2\pi\sqrt{-1}/\sigma}$, in spite of the roundoff error that this introduces in any realistic computer—the experimental results show that the roundoff error causes no apparent loss of validity of the theoretical predictions (i.e., they confirm the theory).

- The algorithm is randomized but its behaviour neither depends on any a priori probabilistic assumption on the input, nor on the size of the alphabet. It proceeds by computing $k$ independent equally distributed estimates for the vector score, and by averaging them. The expected value of the averaged estimator is equal to the exact value. More precisely, we compute our estimate of the score vector $\widehat{C}$ in deterministic time $O(kN\log M)$, such that the expected value of its $i$th component $\widehat{c}_i$ equals $c_i$. Moreover, the standard deviation is bounded above by $(M-c_i)/\sqrt{k}$. Note that we have a trade-off between time complexity and accuracy: by choosing larger values of $k$, more accurate estimates are obtained. Also note that the standard deviation is particularly small when $c_i$ is close to $M$, which is precisely the

case of *almost occurrence* that usually interests us. However, small values for $k$ are in practice sufficient to achieve a reasonable accuracy.

We summarize our main results in the following theorem.

**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 $(M - c_i)^2/k$.*

Although we feel that the algorithm should work best with the FFT step performed by dedicated chips, we have a full implementation including a soft FFT. Experimental results confirm the theory, and are described in the last section.

## 2 Preliminaries and Terminology

We first observe that it suffices to obtain an algorithm of time complexity $O(kM \log M)$ for the case $N = 2M$. Indeed, if $N > 2M$, we can use the standard technique [7] of partitioning the text into $O(N/M)$ overlapping chunks of length $2M$ each, and then processing each chunk separately in time $O(kM \log M)$. The overall complexity is then $O(kN \log M)$. Therefore we henceforth assume, without loss of generality, that $N = 2M$.

Let $\mathcal{A}$ be a finite alphabet of cardinality $\sigma$. We use the following notation for the integer interval

$$[0, \sigma[ = \{0, \cdots, \sigma - 1\}.$$

We henceforth use $\omega$ to denote any primitive $\sigma$th root of unity and $\Sigma$ to denote the set of all possible mappings from $\mathcal{A}$ to $[0, \sigma[$. Observe that, for a random variable $X$ that is uniformly distributed over $[0, \sigma[$, we have $\mathrm{E}\left(\omega^X\right) = 0$. This fact stems from the nullity of the sum of all the $\sigma$th roots of unity. As a corollary, if the random variable $\Phi$ is uniformly distributed over $\Sigma$, then $\mathrm{E}\left(\omega^{\Phi(a)}\right) = 0$ for any $a \in \mathcal{A}$.

## 3 The Algorithm

The key idea of our algorithm is to iteratively compute randomized estimated values for $C$; the values obtained are those of a random variable whose mean is the score and whose variance is small. We then repeat the calculation to estimate the score with good probability. In the sequel, $k$ denotes a positive integer, the number of repetitions of the algorithm. As will be shown in the next section, the larger $k$, the smaller the variance of our answers.

The iteration step of the algorithm is based on the following idea: assume that we have two strings of length $M$; if we renumber the letters at random with numbers from $[0, \sigma[$, we obtain two integer sequences $n_0 \ldots n_{M-1}$ and $m_0 \ldots m_{M-1}$; in the mean over all renumberings, the Hermitian inner product

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

counts the number of matches between both strings. We therefore have the following algorithm to compute the score, named MC after the Monte-Carlo approach used:

**Algorithm MC**

INPUT: a text $T = t_0 \dots t_{2M-1}$ and a pattern $P = p_0 \dots p_{M-1}$ where the $t_i$'s and the $p_i$'s are letters from $\mathcal{A}$;
OUTPUT: an estimate for the score vector $C$.

1. For $\ell = 1, 2, \cdots, k$:
   (a) select randomly and uniformly a $\Phi^{(\ell)}$ from $\Sigma$;
   (b) from the text $T$, obtain a complex sequence $T^{(\ell)}$ of size $2M$ 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 $M$ (trailing) zeroes;
   (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 our estimate of $C$.

The previous algorithm deserves several remarks.

- It is crucial that $\Phi^{(\ell)}$ be a random *mapping* rather than a random *permutation*. In fact, developing the analysis of the next section with a permutation rather than a mapping would reveal that the corresponding estimate is biased, whereas we need an unbiased estimate of $C$.
- The computation of the convolution is performed by fast Fourier transform. The time complexity of this classical algorithm is $O(M \log M)$, which makes us achieve a low complexity for the overall algorithm. Besides, this algorithm is now implemented in dedicated chips.
- The fast Fourier transform evaluates polynomials related to its inputs at roots of unity. The latter are not related to $\omega$; their order is not $\sigma$ but the size of the text $2M$.
- Of course, when implementing the algorithm, one should reuse the same array for all the $C^{(\ell)}$'s, so as to achieve a space complexity of $O(M)$ rather than $O(kM)$. (The time complexity is left unchanged by this optimization.)

# 4 Analysis

The analysis of this section will show that $\mathrm{E}\left(\widehat{C}\right) = C$, and that the standard deviation of $\widehat{c}_i$ is bounded above by $(M - c_i)/\sqrt{k}$. One is usually only interested in the positions $i$ at which $c_i$ is very close to $M$ (i.e., where the pattern almost occurs in the text). A fact of interest is that it is precisely for these $i$'s that the standard deviation is the smallest. Therefore $\widehat{c}_i$ is a particularly good estimator of $c_i$ when $c_i$ is close to $M$. Here by *close to* $M$ we mean a high percentage of agreement, i.e., $c_i = \lambda M$ where $\lambda$ is a constant close to 1.

We proceed to estimate the mean and the variance of the $\widehat{c_i}$'s. All the random variables $\widehat{c_i}$ are defined in a similar way; hence we generically consider the random variable

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

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

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

where once again $\delta_{x,y}$ denotes the Kronecker symbol.

The random variable $\widehat{s}$ 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 $\widehat{s}$ are then

$$\mathrm{E}(\widehat{s}) = \mathrm{E}(s) \qquad \text{and} \qquad \mathrm{Var}(\widehat{s}) = \frac{\mathrm{Var}(s)}{k}.$$

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

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

**Proof.** The mean of $\widehat{s}$ is

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

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 and the sum of all the $\sigma$th roots of unity is zero. More precisely, we have the equality

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

from which the result follows. □

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

**Lemma 2** *The variance of $\widehat{s}$ is bounded as follows:*

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

Theorem 1 now follows from Lemmata 1 and 2.

# 5 Generalizations

In this section, we extend the previous technique to several directions; our main contribution here is to show that classical generalizations also apply to our algorithm and to perform the corresponding complexity analyzes. We first introduce and analyze 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, we extend our algorithm to accomodate 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, since the former is obtained by setting $f(a) = 1$ and $g(a) = w(a)$ in the latter, and since the latter is conversely obtained by setting $w(a) = f(a)g(a)$ in the former. 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 $\mathcal{A}^2$. We do this essentially for the purpose of the 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

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

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 $\mathcal{A}$ to $[0, \sigma[$. The weighted score between $t_0 \dots t_{M-1}$ and $p_0 \dots p_{M-1}$ is

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

where once again $\delta_{x,y}$ denotes the Kronecker symbol.

The random variable $\widehat{s}$ 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 the mean and variance of $\widehat{s}$ are then

$$\mathrm{E}(\widehat{s}) = \mathrm{E}(s) \qquad \text{and} \qquad \mathrm{Var}(\widehat{s}) = \frac{\mathrm{Var}(s)}{k}.$$

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 $\widehat{s}$ is the weighted score*

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

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

**Proof.** The mean of $\widehat{s}$ is

$$\mathrm{E}(\widehat{s}) = \mathrm{E}(s) = \sum_{j=0}^{M-1} \mathrm{E}\left(h(t_j, p_j)\omega^{\Phi(t_j)-\Phi(p_j)}\right) = \sum_{j=0}^{M-1} h(t_j, p_j)\mathrm{E}\left(\omega^{\Phi(t_j)-\Phi(p_j)}\right) = c,$$

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

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

**Lemma 4** *The variance of $\widehat{s}$ is bounded by*

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

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

**Proof.** Since $\mathrm{Var}(s) = \mathrm{E}(|s|^2) - |\mathrm{E}(s)|^2$, we first study the mean of $|s|^2 = s\overline{s}$. It is

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

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 $\mathrm{E}\left(\omega^{\Phi(t_i)-\Phi(p_i)-\Phi(t_j)+\Phi(p_j)}\right)$ is 1; otherwise, it is 0. The use of random applications instead of random permutations is crucial here to obtain this unbiased estimator: using

permutations, we would still get 1 in the mean when the power is 1 independently from $\Phi$, but not necessarily 0 in the other cases.

By a simple inclusion-exclusion argument, it follows that

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

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

$$\mathrm{Var}(s) = \mathrm{E}\left(|s|^2\right) - |\mathrm{E}(s)|^2 = \sum_{0 \leq i,j < M} h(t_i,p_i)\overline{h(t_j,p_j)}\delta_{t_i,t_j}\delta_{p_i,p_j}\left(1 - \delta_{t_i,p_i}\delta_{t_j,p_j}\right).$$

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

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

and the vector $H$ with $i$th entry $h(t_i,p_i)$. We obtain $\mathrm{Var}(s) = \overline{H}^T\Gamma H$, 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 non-negative and $\rho(\Gamma)$ is the largest eigenvalue. We have

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

To improve on the latter 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 $t_i = t_j \neq p_i = p_j$. It entails that in case of a match $t_i = p_i$, both the $i$th line and the $i$th column of $\Gamma$ are 0. After renumbering the lines and columns in $\Gamma$ and $H$, we part the latter as follows:

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

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

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

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

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

Furthermore, setting

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

we obtain

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

Finally,

$$\mathrm{Var}\left(\widehat{s}\right) = \frac{\mathrm{Var}(s)}{k} = \frac{\overline{H}^T\Gamma H}{k} = \frac{\overline{H'}^T\Gamma' H'}{k} \leq \frac{||h||_\infty^2 (M-c)^2}{k}.$$

□

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*

$$\mathrm{E}\left(\widehat{C}\right) = C \qquad \text{and} \qquad \mathrm{Var}(\widehat{c}_i) \leq \frac{||h||_\infty^2 (M - c_i)^2}{k}.$$

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

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

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 non-weighted 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 the sequel 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 $\mathrm{E}\left(\widehat{C}\right) = C$ 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 [7] 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 *begins* (i.e., has its top-left corner) at position $(mi, mj)$ 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 $2m \times m$, $T_{i,j}$ 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 [7], and in the process introduces "never match" symbols: that is, if $\mathcal{A}$ is the alphabet for the two-dimensional problem, then the corresponding alphabet for the one-dimensional problem is $\mathcal{A} \cup \{\#\}$ 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 $0 \leq i \leq 2m^2$ 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 much more complicated by the presence of the new, special # symbol: we simply follow the rules of Algorithm MC except that, at the place where the algorithm requires 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 $a \in \mathcal{A}$ 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*

$$\mathrm{E}\left(\widehat{C}\right) = C \qquad \text{and} \qquad \mathrm{Var}\left(\widehat{c_{i,j}}\right) \leq \frac{(M - c_{i,j})^2}{k}.$$

**String Matching with Classes.** Let $a_v$ be letters in an alphabet. By a *class* $[a_1 \ldots a_r]$ 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 $P = p_0 \ldots p_{M-1}$ is now a class $[p_{j,1} \ldots p_{j,r_j}]$. 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 $p_{j,1} \ldots p_{j,r_j}$ 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. We then obtain 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 $p = p_0 \ldots p_{M-1}$ of length $M$ for classes $p_j = [p_{j,0} \ldots p_{j,r_j}]$ can be computed by a Monte-Carlo algorithm in deterministic time $O(kN \log M)$ with mean and variance*

$$\mathrm{E}\left(\widehat{C}\right) = C \qquad \text{and} \qquad \mathrm{Var}(\widehat{c_i}) \leq \frac{||h||_\infty^2 (M' - c_i)^2}{k} \qquad \text{for} \qquad M' = \sum_{j=1}^{M} r_j \geq M.$$

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: let us 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

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

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 our 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 pattern matching on a given alphabet $\mathcal{A}$ and achieve a better accuracy of the estimates, we adjoin two new special symbols, a "never match" symbol # and an "always match" symbol $*$.

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 $a \in \mathcal{A}$. It may simultaneously be used 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 $\mathcal{A} \cup \{\#\}$ does not change the analysis of the previous sections.

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

$$f(*) = 1 \qquad \text{and} \qquad g(*) = w(*).$$

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 1's and mismatches by 0's ($w = 1$). As a convention, the "always match" symbol matches itself; whether "always match" and "never match" symbols match each other is almost always irrelevant in the sequel. In all the cases, we get an algorithm that is only four times slower to yield sharper estimates. This algorithm 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).

**Algorithm MC with "Always Match" Symbols**

INPUT: a text $T = t_0 \ldots t_{2M-1}$ and a pattern $P = p_0 \ldots p_{M-1}$ where the $t_i$'s and the $p_i$'s are letters from $\mathcal{A} \cup \{*\}$;
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.

As an optimisation, 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 "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, 3 and 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*

$$\mathrm{E}\left(\widehat{C}\right) = C \qquad \text{and} \qquad \mathrm{Var}(\widehat{c}_i) \leq \frac{||h||_\infty^2 (M - c_i)^2}{k}.$$

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 $i$'s. 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.

**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 $[a_1 \ldots a_r]$, we introduce a new symbol $\overline{[a_1 \ldots a_r]}$ 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 non-weighted alphabet. In this case, the match of a letter $b$ against the class complement $\overline{[a_1 \ldots a_r]}$ is counted by the match of $b$ against $*$ minus the match of $b$ against $[a_1 \ldots a_r]$. This yields the following algorithm, where class complements are basically viewed as "always match" symbol and a correcting contribution is removed at Step 1.

**Algorithm MC with Class Complements**

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

1. Replace all $*$ by # and each class complement $\overline{[a_1 \ldots a_r]}$ 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.

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

# 6 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. What is observed is in excellent agreement of the experiments with the phenomena predicted by the theory. The algorithm behaves well in practice: it 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 percent of estimated mismatch.

Our experiments concern the unweighted model only, although we allowed "never match" symbols. On each experiment, we have performed a comparison between three algorithms: a naive algorithm, the shift-add algorithm due to Baeza-Yates and Gonnet [5], our Monte-Carlo algorithm. The naive algorithm consists in two nested loops with an inner test to take "never match" symbols into account. Concerning the shift-add algorithm, we have not packed the state of the search on a single machine word, since this is not possible for $M$ larger than, say, 64. In [5], Baeza-Yates and Gonnet deal with "don't care" symbols that are "always match" symbols. To take our "never match" symbols into account, we used a straightforward variation of the algorithm inspired by the treatment of "don't care" symbols in [5]. For a fair comparison, all the implementations use the same I/O routines, and are designed to compute vector scores, i.e., they do not only detect the positions with a number of mismatches less than a given threshold. In particular, for the Baeza-Yates and Gonnet algorithm, we count mismatches up to the size $M$ of the pattern and not up to a fixed number independent from $M$. However, our Monte-Carlo implementation was set up so as to compute the approximate score vector only, while both other algorithms compute exact scores. Using the result of our algorithm as a filter, one can then compute exact scores at each positions where the estimated score reaches a given threshold. The timings of this postprocessing which correspond to a high threshold (0.8 or more) are very low, while those corresponding to a low threshold (zero or less) are those of the naive algorithm. We therefore do not report these timings in the description of the experiments below.

In the sequel, the algorithms are referred to as Naive, SA and MC, respectively.

**Randomly Generated Text.** A first experiment was performed with a text of 8192 bytes chosen at random according to the uniform distribution over an alphabet of size 256. 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. The results available in Figure 2 show that our Monte-Carlo is very efficient with large patterns.

| Test | Naive | SA | MC ($k = 1$) | MC ($k = 2$) | MC ($k = 3$) |
|---|---|---|---|---|---|
| Time (s) | 7.8 | 15.4 | 1.8 | 3.5 | 5.1 |
| Flow (kB/s) | 1.0 | 0.5 | 4.5 | 2.3 | 1.6 |

Figure 2: Comparison of the quadratic algorithm (Naive), the Baeza-Yates and Gonnet algorithm (SA) and our Monte-Carlo algorithm (MC) on a 4096-byte pattern and a randomly generated text.

With a different setting of our Monte-Carlo algorithm, all the estimated scores were returned together with the corresponding exact scores. Apart from the almost complete match with score 4042, all other positions have a score less than or equal to 59. The almost complete match is detected by our algorithm, with an estimated score with an error of less than 0.2 percent, while the program behaves well on all other shifts, with scores that were not overestimated by more than a factor of 5.

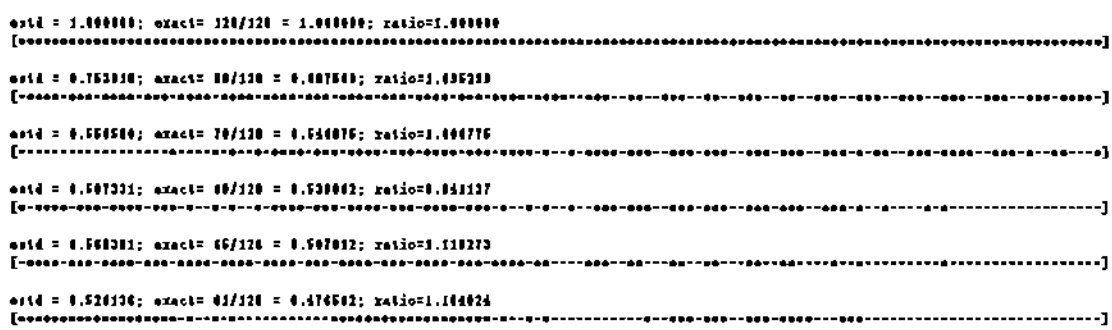
**Classical Music.** As another example, we considered the search for approximate occurrences of a clarinet theme of Beethoven's Fifth Symphony. Although our algorithm is slower than both other ones for small texts and patterns, we show by this experiment that it is still very good at locating interesting events.

For the experiment, the datas are MIDI code, and the length of the theme is $M = 128$, so that we set $N = 2M = 256$, $\sigma = 128$ and we performed the algorithm for $k = 1$, 2 and 3. The timings are in Figure 3.

| Test | Naive | SA | MC ($k = 1$) | MC ($k = 2$) | MC ($k = 3$) |
|---|---|---|---|---|---|
| Time (s) | 2.4 | 4.2 | 9.0 | 17.4 | 26.5 |
| Flow (kB/s) | 44.7 | 25.6 | 11.9 | 6.2 | 4.1 |

Figure 3: Comparison of the quadratic algorithm (Naive), the Baeza-Yates and Gonnet algorithm (SA) and our Monte-Carlo algorithm (MC) on a 128-byte pattern and MIDI encoding of classical music.

With another version of our Monte-Carlo algorithm, a threshold of $\lambda = 0.5$ was used in order to filter out approximate matches (i.e., the program outputs only those matches with $c \geq \lambda M$). Here are selected parts of the output for $k = 3$, sorted by decreasing scores:



Each block corresponds to a certain position. In 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 $\widehat{c}_i/c_i$. The characters * 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). Note the accuracy of the algorithm on this execution: the ratio $\widehat{c}_i/c_i$ varies little around 1.

**Sequenced Proteins.** In view of the previous results, a natural problem is to determine for which values $M$ of the size of the patterns our Monte-Carlo algorithm is practically faster than the other ones. To this end, we have performed the search of prefixes of a sequenced protein in a database of sequenced proteins. Although the proteins are encoded over an alphabet with a few dozens of characters only, the database also contains various information and comments, so that we kept the alphabet consisting of all ASCII codes. Beside this, we allowed "never match" symbols both in the patterns and in the text, since the protein database we used contains such "never match" symbols. More specifically, the parameters for these experiments are $N = 2313316$, $\sigma = 128$ and $k = 2$. The patterns used are of size $M = 2^p$ for $p$ from 7 to 12 (from $M = 256$ to $M = 4096$). The timing results are reproduced in Figure 4. Again, the estimations of the scores are accurate enough to detect occurrences with few mismatches.

| M | Naive | SA | MC | Naive | SA | MC |
|---|---|---|---|---|---|---|
| | Time (s) | | | Flow (kB/s) | | |
| 128 | 49.5 | 88.9 | 368.8 | 45.6 | 25.4 | 6.1 |
| 256 | 94.7 | 171.8 | 407.6 | 23.9 | 13.2 | 5.5 |
| 512 | 187.5 | 332.2 | 463.9 | 12.0 | 6.8 | 4.9 |
| 1024 | 368.8 | 682.0 | 513.3 | 6.1 | 3.3 | 4.4 |
| 2048 | 847.1 | 1356.0 | 558.8 | 2.7 | 1.7 | 4.0 |
| 4096 | 1828.6 | 2749.4 | 614.9 | 1.2 | 0.8 | 3.7 |

Figure 4: Comparison of the quadratic algorithm (Naive), the Baeza-Yates and Gonnet algorithm (SA) and our Monte-Carlo algorithm (MC) on a 4096-byte pattern and a randomly generated text.

Using regression methods, we get the following empirical formulae for the practical time complexity of the algorithms:

$$t_{\text{Naive}} = -42.7 + 0.450 \times M, \quad t_{\text{SA}} = -4.79 + 0.671 \times M \quad \text{and} \quad t_{\text{MC}} = 17.4 + 49.5 \times \log_2 M.$$

These formulae interpolate the empirical values with high accuracy, which makes our experiments confirm the theory. Next, we get thresholds from these formulae: our Monte-Carlo algorithm gets faster than the shift-add algorithm for $M > 736$, next faster than the naive algorithm for $M > 1267$.

**Conclusion.** Although the purpose of our experiments was to investigate the quality of the approximation rather than the speed of the algorithm, we can make a few comments about the latter. In all our experiments the shift-add method is roughly twice slower than the naive algorithm. As already explained, this is due to the values of $M$ that we used: for small $M$, the shift-add methods gets three times faster than the naive algorithm [5], because several parameters of the algorithm can be packed in the same machine word and processed simultaneously. The threshold given in [5]

is 9 for 32-bit machines. On the other hand, our method based on FFT is typically faster than the naive method when $M \geq 2048$. As mentioned earlier, we used a soft implementation of FFT (which suffers from large constant factors in its time complexity), and our algorithm should work better with the FFT step performed by dedicated chips, inducing a much lower threshold than the current one of 2048.

**Acknowledgement.** The authors warmly thank Roberto Sierra (bert@netcom.com) who sequenced the whole symphony in MIDI code, together with other musical pieces, and made them freely available on the WEB.

The authors are grateful to an anonymous referee for suggesting an experimental comparison of our algorithm to Baeza-Yates' and Gonnet's shift-add algorithm.

## References

[1] A. Apostolico and Z. Galil (Eds), *Combinatorial Algorithms on Words*, Springer, 1985.

[2] K. Abrahamson, "Generalized String Matching," *SIAM Journal of Computing*, 16, 1987, pp. 1039–1051.

[3] M.J. Atallah, Y. Génin, and W. Szpankowski, "A Pattern Matching Approach to Image Compression," *Proceedings of the Third IEEE International Conference on Image Processing*, Lausanne, Switzerland, 1996, pp. 349–356.

[4] M.J. Atallah, P. Jacquet, and W. Szpankowski, "A Probabilistic Approach to Pattern Matching with Mismatches," *Random Structures and Algorithms*, 4, 1993, pp. 191–213.

[5] R.A. Baeza-Yates and G.H. Gonnet, "A New Approach to Text Searching," *Communications of the ACM*, 35, 1992, pp. 74–82.

[6] R.A. Baeza-Yates and C.H. Perleberg, "Fast and Practical Approximate Pattern Matching," *Information Processing Letters*, 59, 1996, pp. 21–27.

[7] M. Crochemore and W. Rytter, *Text Algorithms*, Oxford University Press, 1994.

[8] M.J. Fischer and M.S. Paterson, "String Matching and Other Products," *Complexity of Computation, SIAM-AMS Proceedings*, 7, 1974, pp. 113–125.

[9] H. Karloff, "Fast Algorithms for Approximately Counting Mismatches," *Information Processing Letters*, 48, 1993, pp. 53–60.

[10] S.R. Kosaraju, "Efficient String Matching," manuscript, Johns Hopkins University, 1987.

[11] S. Kumar and E.H. Spafford, "A Pattern-Matching Model for Intrusion Detection," *Proceedings of the National Computer Security Conference*, 1994, pp. 11–21.

[12] D.E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, Addison-Wesley, 2nd ed., 1981, pp. 290–294.