Bit-parallel (δ, γ)-matching and suffix automata ☆

Maxime Crochemore [a,b,1], Costas S. Iliopoulos [b],
Gonzalo Navarro [c,2,3,*], Yoan J. Pinzon [b,d,2], Alejandro Salinger [c]

[a] *Institut Gaspard-Monge, Université de Marne-la-Vallée, France*
[b] *Department of Computer Science, King's College, London, England, UK*
[c] *Center for Web Research, Department of Computer Science, University of Chile, Chile*
[d] *Lab. de Cómputo Especializado, Univ. Autónoma de Bucaramanga, Colombia*

Abstract

(δ, γ)-matching is a string matching problem with applications to music retrieval. The goal is, given a pattern $P_{1\ldots m}$ and a text $T_{1\ldots n}$ on an alphabet of integers, find the occurrences P' of the pattern in the text such that (i) $\forall 1 \leqslant i \leqslant m$, $|P_i - P'_i| \leqslant \delta$, and (ii) $\sum_{1 \leqslant i \leqslant m} |P_i - P'_i| \leqslant \gamma$. The problem makes sense for $\delta \leqslant \gamma \leqslant \delta m$. Several techniques for (δ, γ)-matching have been proposed, based on bit-parallelism or on skipping characters. We first present an $\mathrm{O}(mn \log(\gamma)/w)$ worst-case time and $\mathrm{O}(n)$ average-case time bit-parallel algorithm (being w the number of bits in the computer word). It improves the previous $\mathrm{O}(mn \log(\delta m)/w)$ worst-case time algorithm of the same type. Second, we combine our bit-parallel algorithm with suffix automata to obtain the first algorithm that skips characters using both δ and γ. This algorithm examines less characters than any previous approach, as the others do just δ-matching and check the γ-condition on the candidates. We implemented our algorithms and drew experimental results on real music, showing that our algorithms are superior to current alternatives with high values of δ.

1. Introduction

The string matching problem is to find all the occurrences of a given pattern $P_{1...m}$ in a large text $T_{1...n}$, both being sequences of characters drawn from a finite character set Σ. This problem is fundamental in computer science and is a basic need of many applications, such as text retrieval, music retrieval, computational biology, data mining, network security, etc. Several of these applications require, however, more sophisticated forms of searching, in the sense of extending the basic paradigm of the pattern being a simple sequence of characters.

In this paper we are interested in music retrieval. A musical score can be viewed as a string: at a very rudimentary level, the alphabet could simply be the set of notes in the chromatic or diatonic notation, or the set of intervals that appear between notes (e.g., pitch may be represented as MIDI numbers and pitch intervals as number of semitones). It is known that exact matching cannot be used to find occurrences of a particular melody, so one resorts to different forms of *approximate* matching, where a limited amount of *differences* of diverse kinds are permitted between the search pattern and its occurrence in the text.

The approximate matching problem has been used for a variety of musical applications [6,9,15,19,20]. Most computer-aided musical applications adopt an absolute numeric pitch representation (most commonly MIDI pitch and pitch intervals in semitones; duration is also encoded in numeric form). The absolute pitch encoding, however, may be insufficient for applications in tonal music as it disregards tonal qualities of pitches and pitch-intervals (e.g., a tonal transposition from a major to a minor key results in a different encoding of the musical passage and thus exact matching cannot detect the similarity between the two passages). One way to account for similarity between closely related but non-identical musical strings is to permit a difference of at most δ units between the pattern character and its corresponding text character in an occurrence, e.g., a C-major $\{60, 64, 65, 67\}$ and a C-minor $\{60, 63, 65, 67\}$ sequence can be matched if a tolerance $\delta = 1$ is allowed in the matching process. Additionally, we require that the total number of differences across all the pattern positions does not exceed γ, in order to limit the total number of differences while keeping sufficient flexibility at individual positions.

The formalization of the above problem is called (δ, γ)-matching. The problem is defined as follows: the alphabet Σ is assumed to be a set of integer numbers, $\Sigma \subset \mathbb{Z}$. Apart from the pattern P and the text T, two extra parameters, $\delta, \gamma \in \mathbb{N}$, are given. The goal is to find all the occurrences P' of P in T such that (i) $\forall 1 \leqslant i \leqslant m$, $|P_i - P'_i| \leqslant \delta$, and (ii) $\sum_{1 \leqslant i \leqslant m} |P_i - P'_i| \leqslant \gamma$. Note that the problem makes sense for $\delta \leqslant \gamma \leqslant \delta m$: If $\gamma > \delta$ then the limit on the sum of differences is larger than the limit on any difference, so one should set $\delta \leftarrow \gamma$; and if $\gamma > \delta m$ then condition (i) implies (ii) and we should set $\gamma \leftarrow \delta m$.

Several recent algorithms exist to solve this problem. These can be classified as follows:

Bit-parallel: The idea is to take advantage of the intrinsic parallelism of the bit operations inside a computer word of w bits [1], so as to pack several values in a single word and manage to update them all in one shot. In [7,8] this approach was used to obtain SHIFT-PLUS, an $O(nm \log(\delta m)/w)$ worst-case time algorithm. The algorithm packs m counters whose maximum value is $m\delta$, hence it needs $m \lceil \log_2(\delta m + 1) \rceil$ bits overall and $O(m \log(\delta m)/w)$ computer words have to be updated for each text character.

Occurrence heuristics: Inspired by Boyer–Moore techniques [5,21], they skip some text characters according to the position of some characters in the pattern. In [7], several algorithms of this type were proposed for δ-matching (a restricted case where $\gamma = \delta m$), and they were extended to general (δ, γ)-matching in [10]. The extension is done by checking the γ-condition on each candidate that δ-matches the pattern. These algorithms are TUNED-BOYER–MOORE, SKIP-SEARCH and MAXIMAL-SHIFT, each of which has a counterpart in exact string matching. These algorithms are faster than the bit-parallel ones, as they are simple and skip text characters.

Substring heuristics: Based on suffix automata [12,13], these algorithms skip text characters according to the position of some pattern substrings. In [10,11], three algorithms of this type, called δ-BM1, δ-BM2 and δ-BM3, are proposed. They try to generalize the suffix automata to δ-matching, but they obtain only an approximation that accepts more occurrences than necessary, and these have to be verified later. They also verify the γ-condition over each δ-matching candidate.

In this paper we present two new (δ, γ)-matching algorithms:

- We improve SHIFT-PLUS in two aspects. First, we show that its worst case complexity can be reduced to $O(nm \log(\gamma)/w)$ by means of a more sophisticated counter management scheme that needs only $\lceil 1 + \log_2(\gamma + 1) \rceil$ bits per counter. Second, we show how its average-case complexity can be reduced to $O(n)$.
- We combine our bit-parallel algorithm with suffix automata, as already done with other string matching problems [17,18], so as to obtain the first algorithm able of skipping text characters based both on δ- and γ- conditions. All previous algorithms skip characters using the δ-condition only. Moreover, our suffix automaton accepts exactly the suffixes of strings that (δ, γ)-match our pattern, so no candidate verification is necessary at all. Our algorithm examines less characters than any previous technique.

The algorithms are very efficient and simple to implement. Our experimental results on real music data show that they improve previous work when δ is large (so that their dependence on γ rather than on δ shows up). For short patterns, of length up to 20, the character skipping algorithm is the best, otherwise our simple bit-parallel algorithm dominates.

2. Basic concepts

In this section we present the concepts our paper builds on: bit-parallelism and suffix automata. We start by introducing some terminology.

A string $x \in \Sigma^*$ is a *factor* (or substring) of P if P can be written $P = uxv$, $u, v \in \Sigma^*$. A factor x of P is called a *suffix* (*prefix*) of P if $P = ux$ ($P = xu$), $u \in \Sigma^*$.

A *bit mask* of length r is simply a sequence of bits, denoted $b_r \ldots b_1$. We use exponentiation to denote bit repetition (e.g., $0^3 1 = 0001$). The length of the computer word is w bits, so the mask of length $r \leqslant w$ is stored somewhere inside the computer word. Also, we write $[x]_r$ to denote the binary representation of number $x < 2^r$ using r bits. We also use C-like syntax for operations on the bits of computer words: "|" is the bitwise-or, "&" is the bitwise-and, and "~" complements all the bits. The shift-left operation, "<<", moves the bits to the left and enters zeros from the right, that is, $b_m b_{m-1} \ldots b_2 b_1 << r = b_{m-r} \ldots b_2 b_1 0^r$. Finally, we can perform arithmetic operations on the bits, such as addition and subtraction, which operate the masks as numbers. For instance, $b_r \ldots b_x 10000 - 1 = b_r \ldots b_x 01111$.

2.1. Bit-parallelism

In [2,23], a new approach to text searching was proposed. It is based on *bit-parallelism* [1], a technique consisting in taking advantage of the intrinsic parallelism of the bit operations inside a computer word. By using cleverly this fact, the number of operations that an algorithm performs can be cut down by a factor of at most w, the number of bits in the computer word. Since in current architectures w is 32 or 64, the speedup is very significant in practice.

The Shift-And algorithm [23] uses bit-parallelism to simulate the operation of a nondeterministic automaton that searches the text for the pattern (see Fig. 1). A plain simulation of that automaton takes time $O(mn)$, and Shift-And achieves $O(mn/w)$ worst-case time (optimal speedup).

The algorithm first builds a table B which for each character $c \in \Sigma$ stores a bit mask $B[c] = b_m \ldots b_1$, so that $b_i = 1$ if and only if $P_i = c$. The state of the search is kept in a bit mask $D = d_m \ldots d_1$, where $d_i = 1$ whenever the state numbered i in Fig. 1 is active. That is, after having scanned text position j, we have $d_i = 1$ whenever $P_{1 \ldots i} = T_{j-i+1 \ldots j}$. Therefore, we report a match whenever d_m is set.

We start with $D = 0^m$ and, for each new text character T_j, update D using the formula

$$D \leftarrow \big((D << 1) \,|\, 0^{m-1}1\big) \,\&\, B[T_j]$$

because each state may be activated by the previous state as long as T_j matches the corresponding arrow. The "$|\,0^{m-1}1$" after the shift corresponds to the self-loop at the beginning
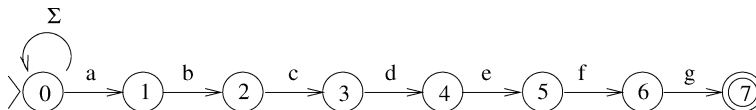

Fig. 1. A nondeterministic automaton to search a text for the pattern $P = $ "abcdefg". The initial state is 0.
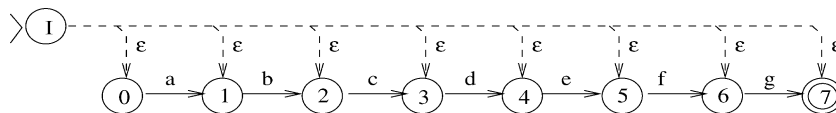
Fig. 2. A nondeterministic suffix automaton for the pattern $P = $ `"abcdefg"`. Dashed lines represent ε-transitions. The initial state is I.

of the automaton (as state 0 is not represented in D). Seen another way, the ith bit is set if and only if the $(i - 1)$th bit was set for the previous text character and the new text character matches the pattern at position i. In other words, $T_{j-i+1...j} = P_{1...i}$ if and only if $T_{j-i+1...j-1} = P_{1...i-1}$ and $T_j = P_i$.

The cost of this algorithm is $O(n)$. For patterns longer than the computer word ($m > w$), the algorithm uses $\lceil m/w \rceil$ computer words for the simulation, with a worst-case cost of $O(mn/w)$. By managing to update only those computer words that have some active state, an average case cost of $O(n)$ is achieved.

It is very easy to extend Shift-And to handle classes of characters, where each pattern position does not match just a single character but a set thereof. If C_i is the set of characters at position i in the pattern, then we set the ith bit of $B[c]$ for all $c \in C_i$.

2.2. Suffix automata

We describe the BDM pattern matching algorithm [12,13], which is based on a suffix automaton. A *suffix automaton* on a pattern $P_{1...m}$ is a deterministic finite automaton that recognizes the suffixes of P. The nondeterministic version of this automaton has a very regular structure (see in Fig. 2).

The (deterministic) suffix automaton is well known [12]. Its size, counting both nodes and edges, is $O(m)$, and it can be built in $O(m)$ time [12]. A very important fact is that this automaton can also be used to recognize the factors of P: The automaton is active as long as we have read a factor of P.

This structure is used in [12,13] to design a pattern matching algorithm called BDM, which is optimal on average ($O(n \log_{|\Sigma|}(m)/m)$ time on uniformly distributed text). To search a text T for P, the suffix automaton of $P^r = P_m P_{m-1} \ldots P_1$ (the pattern read backwards) is built. A window of length m is slid along the text, from left to right. The algorithm reads the window right to left and feeds the suffix automaton with the characters read. During this process, if a final state is reached, this means that the window suffix we have traversed is a prefix of P (because suffixes of P^r are reversed prefixes of P). Then we store the current window position in a variable *last*, possibly overwriting its previous value. The backward window traversal ends in two possible forms:

(1) We fail to recognize a factor, that is, we reach a character σ that does not have a transition in the automaton (see Fig. 3). In this case the window suffix read is not a factor of P and therefore it cannot be contained in any occurrence. We can actually shift the window to the right, aligning its starting position to *last*, which corresponds to the longest prefix of P seen in the window. We cannot miss an occurrence because in that case the suffix automaton would have found its prefix in the window.
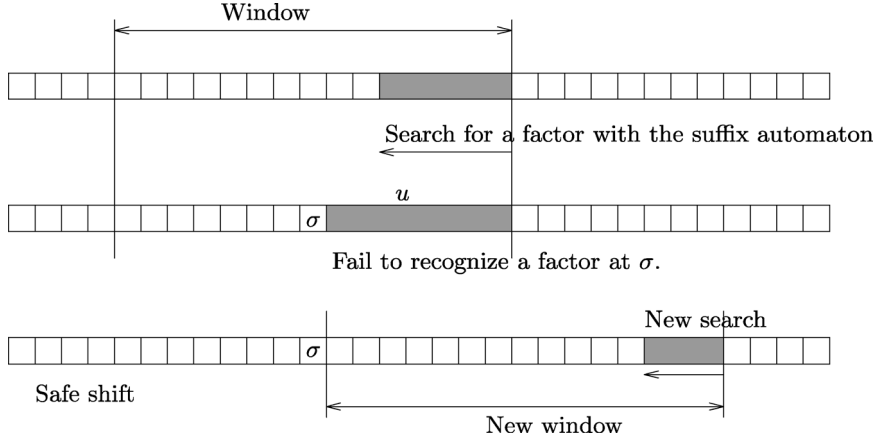
Fig. 3. Basic search with the suffix automaton.

(2) We reach the beginning of the window, therefore recognizing the pattern P. We report the occurrence, and shift the window exactly as in the previous case (we have the previous *last* value).

It is possible to simulate the suffix automaton in nondeterministic form by using bit-parallelism [17,18], so as to obtain very efficient and simple algorithms.

3. Improving the bit-parallel algorithm

First of all, notice that δ-matching is trivial under the bit-parallel approach, as it can be accommodated using the ability to search for classes of characters. We define that pattern character c matches text characters $c - \delta \ldots c + \delta$. Hence, if $B[c] = b_m \ldots b_1$, we set $b_i = 1$ if and only if $|P_i - c| \leqslant \delta$. The rest of the algorithm is unchanged and the same complexities are obtained.

The real challenge is to do (δ, γ)-matching. The solution we present is an improvement over that of [7,8] and it has some resemblances with that of [3] for Hamming distance.

Let us focus for a moment on γ-matching alone. Instead of storing just one bit d_i to tell whether $P_{1\ldots i}$ matches $T_{j-i+1\ldots j}$, we store a counter c_i to record the sum of the absolute differences between the corresponding characters. That is

$$c_i = \sum_{1 \leqslant k \leqslant i} |P_k - T_{j-i+k}| \tag{1}$$

and we wish to report text positions where $c_m \leqslant \gamma$.

The next lemma shows how to update the c_i values for a new text position, and suggests an O(mn) time γ-matching algorithm.

Lemma 1. *Assume that we want to compute the counters $c_1 \ldots c_m$ according to Eq. (1) for text position j, and have computed $c'_1 \ldots c'_m$ for position $j - 1$. The c_i values satisfy*

$$c_i = \sum_{1 \leqslant k \leqslant i} |P_k - T_{j-i+k}| = c'_{i-1} + |P_i - T_j|$$

assuming $c'_0 = 0$.

Proof. Immediate by substitution of c'_{i-1} according to Eq. (1). □

The update technique given in Lemma 1 is good for a bit-parallel approach. Let us assume that each c_i counter will be represented using ℓ bits, where ℓ will be specified later. Hence the state of the search will be expressed using the bit mask

$$D = [c_m]_\ell [c_{m-1}]_\ell \ldots [c_2]_\ell [c_1]_\ell. \tag{2}$$

We precompute a mask $B[c]$ of counters $[b_m]_\ell \ldots [b_1]_\ell$, so that $b_i = |P_i - c|$. Then, the following lemma establishes the bit-parallel formula to update D.

Lemma 2. *Assume that we want to compute bit mask D according to Eq. (2) for text position j, and have computed D' for position $j - 1$. Then*

$$D = (D' << \ell) + B[T_j]. \tag{3}$$

Proof. The ith counter of D' is c'_i. After the shift-left ("$<<$") the ith counter becomes c'_{i-1}. The ith counter of $B[T_j]$ is $|P_i - T_j|$. Hence the ith counter of the right hand side of the equality is $c'_{i-1} + |P_i - T_j|$. According to Lemma 1, this is c_i. □

This gives us a solution for γ-matching. Start with $D = ([\gamma + 1]_\ell)^m$ (to avoid matching before reading T_m) and update it according to Eq. (3). Every time we have $c_m \leqslant \gamma$, report the last text position processed as the end of an occurrence.

In order to include δ-matching in the picture, we change slightly the definition of $B[c]$. The goal is that if, at any position, it holds $|P_i - T_j| > \delta$, then we ensure that the corresponding occurrence is discarded. For this sake, it is enough to redefine $B[c] = [b_m]_\ell \ldots [b_1]_\ell$ as follows:

$$b_i = \textbf{if } |P_i - c| \leqslant \delta \textbf{ then } |P_i - c| \textbf{ else } \gamma + 1. \tag{4}$$

The next lemma establishes the suitability of the above formulas for (δ, γ) matching.

Lemma 3. *If the update formula of Eq. (3) is applied and $B[c]$ is defined according to Eq. (4), then after processing text position j it holds that $c_m \leqslant \gamma$ if and only if $T_{j-m+1\ldots j}$ (δ, γ)-matches P.*

Proof. By Lemmas 1 and 2 and Eq. (1) we have that, if the original definition $b_i = |P_i - c|$ is used, then $c_m = \sum_{1 \leqslant k \leqslant m} |P_k - T_{j-m+k}|$ after processing text position j. The only difference if the definition of Eq. (4) is used is that, if any of the $|P_k - T_{j-m+k}|$ was larger than δ, then $b_k > \gamma$ for $B[T_{j-m+k}]$, and therefore $c_k > \gamma$ after processing text position

$j - m + k$. Since counters only increase as they get shifted and added in Eq. (3), that counter c_k at position $j - m + k$ will become counter c_m at position j, without decreasing. Thus $c_m > \gamma$ after processing text position j. Therefore $c_m \leqslant \gamma$ if and only if $T_{j-m+1\ldots j}$ (δ, γ)-matches P. □

Let us consider now the ℓ value. In principle, using $B[c]$ as in Eq. (4), counter c_m can be as large as $m(\gamma + 1)$, since $b_i \leqslant \gamma + 1$ (recall that $\delta \leqslant \gamma$). However, recall that counter values never decrease as they get shifted over D. This means that, once they become larger than γ, we do not need to know how larger they are. Thus, instead of storing the real c_i value, we would rather store $\min(c_i, \gamma + 1)$, and then need only $\lceil \log_2(\gamma + 2) \rceil$ bits per counter.

In principle, whenever c_i exceeds γ, we store $\gamma + 1$ for it. The problem is how to restore this invariant after adding $B[c]$ to the counters, and also how to avoid overflows in that summation. We show now that we can handle both problems by using the following number of bits per counter:

$$\ell = 1 + \lceil \log_2(\gamma + 1) \rceil. \tag{5}$$

Thus, our bit mask D needs $m\ell = m(1 + \lceil \log_2(\gamma + 1) \rceil)$ bits and our simulation needs $O(m \log(\gamma)/w)$ computer words.

Instead of representing counter c_i as $[c_i]_\ell$, we represent it as

$$c_i \longrightarrow \left[c_i + 2^{\ell-1} - (\gamma + 1) \right]_\ell. \tag{6}$$

This guarantees that the highest bit of the counter will be set if and only if $c_i \geqslant \gamma + 1$, as its representation will be $\geqslant 2^{\ell-1}$.

Before adding $B[T_j]$, we will record all those highest bits in a bit mask $H = D \,\&\, (10^{\ell-1})^m$, and clear those highest bits from D. Once its highest bit is cleared, every counter representation is smaller than $2^{\ell-1}$ and we can safely add b_i without overflowing the counters, since the resulting value is at most $2^{\ell-1} - 1 + (\gamma + 1) = 2^{\ell-1} + \gamma \leqslant 2\gamma + 1$ because of Eq. (5). And again because of Eq. (5), a counter can hold up to value $2(\gamma + 1) - 1 = 2\gamma + 1$. After adding $B[T_j]$ we restore those highest bits set in H.

Note that it is not strictly true that we maintain $\min(c_i, \gamma + 1)$, but it is true that the highest bit of the representation of c_i is set if and only if $c_i > \gamma$, and this is enough for the correctness of the algorithm. The next Lemma establishes this correctness.

Lemma 4. *Assume that c_i is represented as in Eq. (6) if $c_i \leqslant \gamma$, and as $[2^{\ell-1} + x]_\ell$ otherwise, for some $x \geqslant 0$. Then, if the update formula of Lemma 2 is applied with the exception that the highest bits set in the counters are removed before and restored after adding $B[T_j]$, then it holds that the representation is maintained after processing T_j.*

Proof. If c_i already exceeded γ before adding b_i, it will exceed γ after adding b_i. In this case, the representation of c_i was $2^{\ell-1} + x$ and thus it already had its highest bit set. This bit will be restored after adding b_i. Thus, regardless of which value actually stores, the representation will correctly maintain its highest bit set, that is, it will be of the form $2^{\ell-1} + x$ for some $x \geqslant 0$.

Forward-Scan $(P_{1\ldots m}, T_{1\ldots n}, \delta, \gamma)$
```
1.   Preprocessing
2.       ℓ ← 1 + ⌈log₂(γ + 1)⌉
3.       for c ∈ Σ do
4.           B[c] ← ([0]_ℓ)^m
5.           for i ∈ m…1 do
6.               if |c − P_i| ≤ δ then
7.                   B[c] ← (B[c] << ℓ) | (|c − P_i|)
8.               else B[c] ← (B[c] << ℓ) | (γ + 1)
9.   Search
10.      D ← (10^{ℓ−1})^m
11.      for j ∈ 1…n do
12.          if D & 10^{mℓ−1} = 0^{mℓ} then
13.              Report an occurrence at j − m + 1
14.          D ← (D << ℓ) | (2^{ℓ−1} − (γ + 1))
15.          H ← D & (10^{ℓ−1})^m
16.          D ← ((D & ∼ H) + B[T_j]) | H
```

Fig. 4. Bit-parallel algorithm for (δ, γ)-matching. Constant values are precomputed.

On the other hand, if c_i did not exceed γ before adding b_i, then its representation was $c_i + 2^{\ell-1} - (\gamma + 1)$ and the highest bit was not set. Thus the manipulation of highest bits will not affect its result. After the summation the representation will hold $c_i + b_i + 2^{\ell-1} - (\gamma + 1)$. This is a correct representation for the new value $c_i + b_i$, either if $c_i + b_i \leqslant \gamma$ or if $c_i + b_i > \gamma$, as in the latter case the representation is of the form $2^{\ell-1} + x$, where $x = c_i + b_i - (\gamma + 1) \geqslant 0$. □

Fig. 4 depicts the algorithm. It is called *Forward-Scan* to distinguish it from our next algorithms that scan windows backward. The preprocessing consists of computing ℓ according to Eq. (5) and table B according to Eq. (4). Pattern P is processed backwards so as to arrange $B[c]$ in the right order $[b_m]_\ell \ldots [b_1]_\ell$. Line 10 initializes the search by setting $c_i = \gamma + 1$ in D, according to the representation of Eq. (6). Occurrences are reported in lines 12–13, whenever $c_m \leqslant \gamma$, that is, the highest bit of the representation of c_m is not set. Line 14 is the equivalent to $D \leftarrow D << \ell$, except that the counter $c_0 = 0$ that is moved to the position of c_1 must be represented as $2^{\ell-1} - (\gamma + 1)$ according to Eq. (6). Line 15 computes H as explained, to record the highest bits. Line 16 completes the computation of Eq. (3), by removing bits set in H from D and restoring them after the summation with $B[T_j]$.

Assuming that the bit masks fit in a computer word, that is, $m\ell \leqslant w$, the algorithm complexity is $O(m|\Sigma| + n)$. If several computer words are needed, the search complexity becomes $O(mn \log(\gamma)/w)$. However, we defer the details of handling longer bit masks to Section 5, as it is possible to obtain $O(n)$ search time on average.

4. Using suffix automata

As demonstrated in [17,18], the suffix automaton approach of Section 2.2 can be extended to search for more complex patterns by combining it with bit-parallelism. In this

section we combine our bit-parallel approach of Section 3 with the suffix automaton concept to obtain an algorithm that does not inspect all the text characters.

Imagine that we process a text window $T_{pos+1...pos+m}$ right to left. Our goal is that, after having processed T_{pos+j}, we have computed

$$c_i = \sum_{0 \leqslant k \leqslant m-j} |P^r_{i-(m-j)+k} - T_{pos+m-k}|$$

$$= \sum_{0 \leqslant k \leqslant m-j} |P_{2m+1-i-j-k} - T_{pos+m-k}| \qquad (7)$$

for $m - j + 1 \leqslant i \leqslant m$. This can be obtained by initializing $c_i = 0$ before starting processing the window and then updating the c_i values according to the following lemma.

Lemma 5. *Assume that we have values c'_i computed for $T_{pos+j+1}$ according to Eq. (7). Then values c_i for T_{pos+j} satisfy*

$$c_i = c'_{i-1} + |P^r_i - T_{pos+j}|.$$

Proof. It is immediate by rewriting c'_{i-1} according to Eq. (7). □

If we maintain values c_i computed according to Eq. (7), then, after processing T_{pos+j}, (i) if $c_m \leqslant \gamma$, then $\sum_{0 \leqslant k \leqslant m-j} |P_{1+m-j-k} - T_{pos+m-k}| \leqslant \gamma$, that is, $P_{1...m-j+1}$ γ-matches window suffix $T_{pos+j...pos+m}$; (ii) if $c_i > \gamma$ for all $m - j + 1 \leqslant i \leqslant m$, then the window suffix $T_{pos+j...pos+m}$ does not γ-match any pattern substring $P_{m-i+1...(m-i+1)+m-j}$, and therefore no occurrence can contain $T_{pos+j...pos+m}$.

Therefore, a BDM-like algorithm would be as follows. Process text window $T_{pos+1...pos+m}$ by reading it right to left and maintaining c_i values. Every time $c_m \leqslant \gamma$, mark the current window position *last* so as to remember the last time a window suffix γ-matched a pattern prefix. If, at some moment, $c_i > \gamma$ for all i, then shift the window to start at position *last* and restart. If all the window is traversed and still $c_m \leqslant \gamma$, then report the window as an occurrence and also shift it to start at position *last*. The correctness of this scheme should be obvious from Section 2.2.

A bit-parallel computation of the c_i values is very similar to the one developed in Section 3, as the update formulas of Lemmas 1 and 5 are so close. In order to work on P^r, we simply store $B[c]$ in reverse fashion. Vector c_i is initialized at $c_i = 0$ according to Eq. (7). To determine whether $c_m \leqslant \gamma$ we simply test the highest bit of its representation. To determine whether $c_i > \gamma$ for all i we test all highest bits simultaneously. To account also for δ-matching we change the preprocessing of $B[c]$ just as in Eq. (4).

Fig. 5 depicts the algorithm, called "backward-scanning" because of the way windows are processed. The preprocessing is identical to Fig. 4 except that the pattern is processed left to right. D is initialized in line 13 with $c_i = 0$ considering the representation of Eq. (6). Line 14 continues processing the window as long as $c_i \leqslant \gamma$ for some i. The update to D is as in Fig. 4, except that the first shift left ("$<<$") of each window is omitted to avoid losing the first c_1 value. Condition $c_m \leqslant \gamma$ is tested in line 18. When it holds, we update *last* unless we have processed all the window, in which case it means that we found an occurrence and also must maintain the previous *last* value. Line 21 shifts D and introduces

Backward-Scan $(P_{1...m}, T_{1...n}, \delta, \gamma)$

```
1.   Preprocessing
2.       ℓ ← 1 + ⌈log₂(γ + 1)⌉
3.       for c ∈ Σ do
4.           B[c] ← ([0]ℓ)ᵐ
5.           for i ∈ 1...m do
6.               if |c − Pᵢ| ⩽ δ then
7.                   B[c] ← (B[c] << ℓ) | (|c − Pᵢ|)
8.               else B[c] ← (B[c] << ℓ) | (γ + 1)
9.   Search
10.      pos ← 0
11.      while pos ⩽ n − m do
12.          j ← m, last ← m
13.          D ← ([2^(ℓ−1) − (γ + 1)]ℓ)ᵐ
14.          while D & (10^(ℓ−1))ᵐ ≠ (10^(ℓ−1))ᵐ do
15.              H ← D & (10^(ℓ−1))ᵐ
16.              D ← ((D & ∼ H) + B[Tⱼ]) | H
17.              j ← j − 1
18.              if D & 10^(mℓ−1) = 0^(mℓ) then
19.                  if j > 0 then last ← j
20.                  else Report an occurrence at pos + 1
21.              D ← (D << ℓ) | 10^(ℓ−1)
22.      pos ← pos + last
```

Fig. 5. Backward scanning algorithm for (δ, γ)-matching. Constant values are precomputed.

values $\gamma + 1$ from the right, to ensure that the relevant i values are $m - j + 1 \leqslant i \leqslant m$ and that the loop will terminate after m iterations. Finally, the window is shifted by *last*.

Note that, given the invariants we maintain, we can report occurrences without any further verification. Moreover, we shift the window as soon as the window suffix read does not (δ, γ)-match a pattern substring. This is the first character-skipping algorithm with these properties. Previous ones only approximate this property and require verifying candidate occurrences. Consequently, we inspect less characters than previous algorithms.

5. Handling longer patterns

Both algorithms presented are limited by the length of the computer word. They work for $m(1 + \lceil \log(\gamma + 1) \rceil) \leqslant w$. However, in most cases this condition is not fulfilled, so we must handle longer patterns.

5.1. Active computer words

The first idea is just to use as many computer words as needed to represent D. In each computer word we store the maximum amount of counters that fully fit into w bits. So we keep $\kappa = \lfloor w/\ell \rfloor$ counters in each word (except the last one, that may be underfilled), needing $\lceil m/\lfloor w/\ell \rfloor \rceil$ words to represent D. Each time we update D we have to process all the words simulating the bit-parallel operations. With this approach, the forward scanning

takes time $O(nm \log(\gamma)/w)$. We remark that previous forward scanning algorithms [7,8] required $O(nm \log(m\delta)/w)$ time, which is strictly worse than our complexity. The difference is that we have managed to keep the counters below $2(\gamma + 1)$ instead of letting them grow up to $m\delta$. This alternative is called simply "Forward" in the experiments.

A key improvement can be made to the forward scanning algorithm by noticing that it is not necessary to update all the computer words at each iteration. As counter c_i stores the sum of all the differences between characters $P_{1...i}$ and their corresponding characters in the text (Eq. (1)), depending on the values of δ and γ and on the size of the alphabet, most of the time the highest counters will have surpassed γ. Once a counter surpasses γ we only require that it stays larger than γ (recall Section 3 and Lemma 4), so it is not even necessary to update it. Let us say that a computer word is *active* when at least one of its counters stores some $c_i \leqslant \gamma$. The improvement works as follows:

- At each iteration of the algorithm we update the computer words only until the one we have marked as the last active word.
- As we update each word we check whether it is active or not, remembering the new last active word.
- Finally, we check if the last counter of the last active word is $\leqslant \gamma$. In that case, the word that follows the last active word must be the new last active word, as in the next iteration its first counter may become less than $\gamma + 1$, and hence we may need to process it.

This algorithm has the same worst-case complexity of the basic one, but the average case is significantly improved. Consider a random text and pattern following a zero-order model (that is, character probabilities are independent of the neighbor characters). Say that p_s is the probability of the sth character of the alphabet. Then the probability that P_i δ-matches a random text character is $\pi_i = \sum_{P_i - \delta \leqslant s \leqslant P_i + \delta} p_s$. The probability of $P_{1...i}$ δ-matching $T_{j-i+1...j}$ for a random text position j is $w_i = \Pi_{1 \leqslant k \leqslant i} \pi_k$.

The first computer word will be always active; the second will be active only if $P_{1...\kappa}$ matches $T_{j-\kappa+1...j}$; the third will be active only if $P_{1...2\kappa}$ matches $T_{j-2\kappa+1...j}$; and so on. Hence the average number of computer words active at a random text position is at most

$$1 + w_\kappa + w_{2\kappa} + \cdots = \sum_{i \geqslant 0} w_{i\kappa}$$

and this is $O(1)$ provided $\pi = \max_{1 \leqslant i \leqslant m} \pi_i < 1$, as in this case $w_i \leqslant \pi^i$ and the average number of active words is $\sum_{i \geqslant 0} w_{i\kappa} \leqslant \sum_{i \geqslant 0} \pi^{i\kappa} = 1/(1 - \pi^\kappa)$.[4]

Hence, as we update $O(1)$ computer words on average, the average search time is $O(n)$. Note that this holds even without considering γ, which in practice reduces the constant factor. The lower the values of δ and γ are, the better the performance will be. This alternative is called "Forward last" in the experiments.

Yet another improvement can be made to this algorithm by combining it with the basic single-word algorithm. We store the first word used to represent D in a register and run the

---

[4] This holds also if there are $O(1)$ i values such that $\pi_i = 1$.

algorithm using just this word, as long as this one is the only active word. Whenever the second word becomes active, we switch to the multiple word algorithm. We switch back to the basic algorithm whenever the first word becomes the last active word. The use of a register to store the first word yields a better performance, as we have to make less memory accesses. The more time the first word is the only active word, the more significant is the improvement. This alternative is called "Forward register" in the experiments.

Unfortunately this idea cannot be applied to the backward-scanning algorithm, as in this one we will have counters $\leqslant \gamma$ uniformly distributed across all the computer words. This happens because $c_i \leqslant \gamma$ after reading T_{pos+j} if $P_{m-i+1...(m-i+1)+m-j}$ (δ, γ)-matches $T_{pos+j...pos+m}$ (Eq. (7)), and this probability does not necessarily decrease with i (actually it is independent of i on a uniform distribution). The plain multi-word backward scanning algorithm is called simply "Backward" in the experiments.

5.2. Pattern partitioning

Another idea to handle long patterns is to partition them into pieces short enough to be handled with the basic algorithm. Notice that if P (δ, γ)-matches $T_{j-m+1...j}$, and we partition P into j disjoint pieces of length $\lfloor m/j \rfloor$ and $\lceil m/j \rceil$, then at least one piece has to (δ, γ')-match its corresponding substring of $T_{j-m+1...j}$, where $\gamma' = \lfloor \gamma/j \rfloor$. The reason is that, otherwise, each piece adds up at least $\gamma' + 1$ differences, and the total is at least $j(\gamma' + 1) = j(\lfloor \gamma/j \rfloor + 1) > j(\gamma/j) = \gamma$, and then γ-matching is not possible.

Hence we run j (δ, γ')-searches for shorter patterns and check every match of a piece for a complete occurrence. The check is simple and does not even need bit parallelism. Note that if $\delta > \gamma'$, we can actually do (γ', γ')-matching.

We must choose the largest j such that

$$\lceil m/j \rceil \left(1 + \lceil \log_2(\lfloor \gamma/j \rfloor + 1) \rceil\right) \leqslant w$$

and hence we perform $j = O(m \log(\gamma)/w)$ searches. For forward scanning, each such search costs $O(n)$. Piece verification cost is negligible on average. Hence the average search time of this approach is $O(nm \log(\gamma)/w)$, which is not attractive compared to the worst-case search time of the basic approach. However, each of these searches can be made using registers for D, so in practice it could be relevant. It could be also relevant for backward matching, where using D in registers is not possible for long patterns.

Furthermore, the pieces can be grouped and searched for together using so-called "superimposition" [4,16]. By making groups of r pieces each, we perform $\lceil j/r \rceil$ searches. For each search, counter b_i of $B[c]$ will store the minimum difference between c and the ith character of any piece in the group, or $\gamma' + 1$ if none of these differences is smaller than $\gamma' + 1$. Every time we find a match of the whole group we check the occurrence of each of the substrings forming that group. For all the pieces that matched we check the occurrence of the whole pattern at the corresponding position. The greater r is, the less searches we perform, but the more time we spend checking occurrences. The time spent in checking occurrences also increases with δ and γ. Because of this, the optimum r depends on δ, γ and m.

These algorithms are called "Forward superp" and "Backward superp" in the experiments. These include the case $r = 1$, where no superimposition is done.

6. Experimental results

In this section we show experimental evidence comparing the different versions of our algorithms against δ-BM2 [10,11], which is the most efficient alternative (δ, γ)-matching algorithm.

The tests were performed using a Pentium IV, 2 GHz, 512 Mb RAM and 512 Kb cache running Suse Linux with $w = 32$. We used the GNU `gcc` compiler version 2.95.3. Each data point represents the median of 100 trials.

We ran our experiments using real music data obtained from a database of MIDI files of classic music, totalizing 10.5Mb of absolute pitches. We focused on typical parameter values for music searching, namely 2–4 for δ, $1.5m$–$2.0m$ for γ, and 10–200 for m.

The results for forward algorithms are shown in Fig. 6. The variants are called "Forward" (plain multiword forward), "Forward last" (same but updating only up to the last relevant word), "Forward register" (same but switching to single-word mode when possible), and "Forward superp" (partition plus superimposing in the way that gives the best results).

(a) $\delta = 2$ and $\gamma = 1.5m$

(b) $\delta = 4$ and $\gamma = 1.5m$

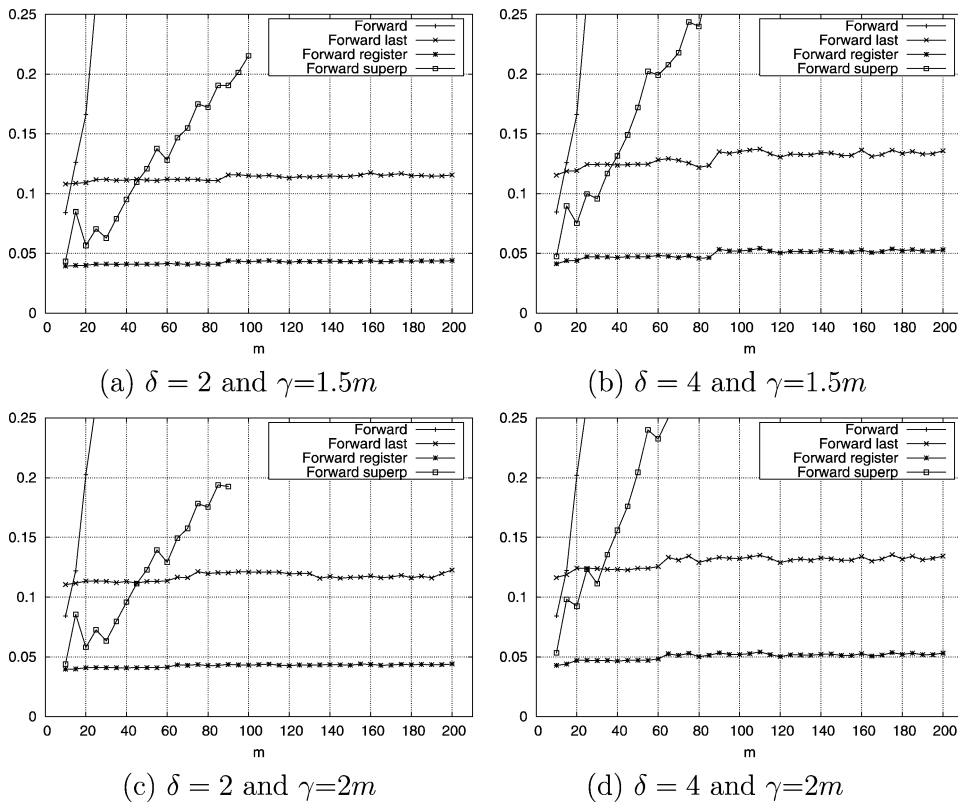(c) $\delta = 2$ and $\gamma = 2m$

(d) $\delta = 4$ and $\gamma = 2m$

Fig. 6. Timing figures for forward algorithms, in seconds per megabyte.

As expected, Forward and Forward-superp are the slowest and their cost grows linearly with m. Forward-superp shows a better constant factor and it is attractive for very short patterns, but soon its linear dependence with m renders it useless. Superimposition alleviates this only partially, as the optimum was to superimpose 2 to 7 patterns for $\delta = 2$ and 2 to 5 for $\delta = 4$. These numbers grow slowly as m increases and stay at a maximum of 5 or 6, making the whole scheme linear in m anyway.

Forward-last and Forward-register, on the other hand, display their O(n) average case time, independent of n. As expected, Forward-register is by far the best. We will consider only this alternative to compare against backward algorithms.

Fig. 7 compares backward algorithms (which includes the relevant competing alternatives), and Forward-register. The backward algorithm only has variants "Backward" (plain single- or multi-word, as needed) and "Backward superp" (partition plus superimposing in the best possible way). δ-BM2 is the best existing alternative algorithm.

We observe that partitioning (including superimposition) is also a bad choice for backward scanning. The reasons are the same as for the forward version. In general, backward searching does not behave competitively when many computer words are involved. Backward was better than Forward-register when the whole (superimposed) representation fit

(a) $\delta = 2$ and $\gamma = 1.5m$

(b) $\delta = 4$ and $\gamma = 1.5m$

(c) $\delta = 2$ and $\gamma = 2m$
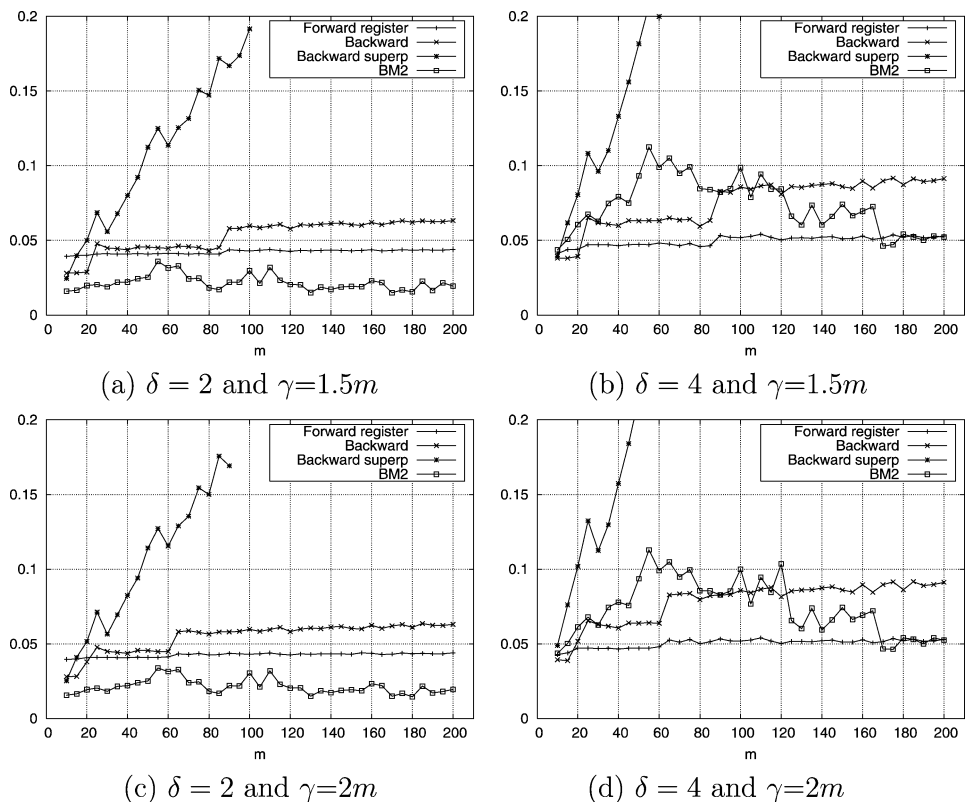
(d) $\delta = 4$ and $\gamma = 2m$

Fig. 7. Timing figures for backward algorithms and the best forward algorithm, in seconds per megabyte.

in a single computer word. As more than a single word is necessary, Forward-register becomes superior. The reason is that backward searching needs to effectively update all its computer words, while the forward versions do so only for a few active computer words.

With respect to the competing algorithm, it can be seen that δ-BM2 is faster than ours for small $\delta = 2$, but as we use a larger $\delta = 4$ it becomes not competitive, as it can be expected from its only-δ filtration scheme. Our algorithms are the only ones that can filter using δ and γ simultaneously.

Finally, we notice that the dependence on δ is significant to the extent that it can double the time it takes by going from $\delta = 2$ to $\delta = 4$. The dependence on γ, on the other hand, is not much significant. We note, however, that Forward-register is rather insensitive to both δ and γ, becoming a strong and stable choice for general (δ, γ)-matching.

7. Conclusions

We have presented new bit-parallel algorithms for (δ, γ)-matching, an extended string matching problem with applications in music retrieval. Our new algorithms make use of bit-parallelism and suffix automata and has several advantages over the previous approaches: they make better use of the bits of the computer word, they inspects less text characters, they are simple, extendible, and robust.

Especially important is that our algorithms are the first truly (δ, γ) character-skipping algorithms, as they skip characters using both criteria. Existing approaches do just δ-matching and check the candidates for the γ-condition. This makes our algorithms a stronger and more stable choice for this problem.

We have also presented several ideas to handle longer patterns, as the algorithms are limited by the length of the computer word. The fastest choice is an algorithm that uses several computer words and updates only those that hold relevant values, switching to single-word mode when possible.

We have shown that our algorithms are the best choice in practice when δ is not small enough to make up a good filter by itself. In this case, the ability of our algorithms to filter with γ at the same time becomes crucial.

We plan to investigate further on more sophisticated matching problems that arise in music retrieval. For example, it would be good to extend (δ, γ)-matching in order to permit insertions and deletions of symbols, as well as transposition invariance. Bit-parallel approaches handling those options, albeit not (δ, γ)-matching at the same time, have recently appeared [14].

Another challenging problem is to consider text indexing, that is, preprocessing the musical strings to speed up searches later. A simple solution is the use of a suffix tree of the text combined with backtracking, which yields search times which are exponential on the pattern length but independent of the text length [22].

References

[1] R. Baeza-Yates, Text retrieval: Theory and practice, in: 12th IFIP World Computer Congress, vol. I, Elsevier Science, 1992, pp. 465–476.

[2] R. Baeza-Yates, G. Gonnet, A new approach to text searching, Comm. ACM 35 (10) (1992) 74–82.

[3] R. Baeza-Yates, G. Gonnet, Fast string matching with mismatches, Inform. and Comput. 108 (2) (1994) 187–199.

[4] R. Baeza-Yates, G. Navarro, Faster approximate string matching, Algorithmica 23 (2) (1999) 127–158.

[5] R.S. Boyer, J.S. Moore, A fast string searching algorithm, Comm. ACM 20 (10) (1977) 762–772.

[6] E. Cambouropoulos, T. Crawford, C. Iliopoulos, Pattern processing in melodic sequences: Challenges, caveats and prospects, in: Proc. Artificial Intelligence and Simulation of Behaviour (AISB'99) Convention, 1999, pp. 42–47.

[7] E. Cambouropoulos, M. Crochemore, C. Iliopoulos, L. Mouchard, Y.J. Pinzon, Algorithms for computing approximate repetitions in musical sequences, in: Proc. 10th Australasian Workshop on Combinatorial Algorithms (AWOCA'99), 1999, pp. 129–144.

[8] E. Cambouropoulos, M. Crochemore, C.S. Iliopoulos, L. Mouchard, Y.J. Pinzon, Algorithms for computing approximate repetitions in musical sequences, Internat. J. Comput. Math. 79 (11) (2002) 1135–1148.

[9] T. Crawford, C. Iliopoulos, R. Raman, String matching techniques for musical similarity and melodic recognition, Computing in Musicology 11 (1998) 73–100.

[10] M. Crochemore, C. Iliopoulos, T. Lecroq, Y.J. Pinzon, W. Plandowski, W. Rytter, Occurrence and substring heuristics for δ-matching, Fundamenta Informaticae 55 (2003) 1–15.

[11] M. Crochemore, C. Iliopoulos, T. Lecroq, W. Plandowski, W. Rytter, Three heuristics for δ-matching: δ-BM algorithms, in: Proc. 13th Ann. Symp. on Combinatorial Pattern Matching (CPM'02), in: Lecture Notes in Comput. Sci., vol. 2373, 2002, pp. 178–189.

[12] M. Crochemore, W. Rytter, Text Algorithms, Oxford University Press, 1994.

[13] A. Czumaj, M. Crochemore, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski, W. Rytter, Speeding up two string-matching algorithms, Algorithmica 12 (1994) 247–267.

[14] K. Lemström, G. Navarro, Flexible and efficient bit-parallel techniques for transposition invariant approximate matching in music retrieval, in: Proc. 10th Internat. Symp. on String Processing and Information Retrieval (SPIRE'03), in: Lecture Notes in Comput. Sci., vol. 2857, 2003, pp. 224–237.

[15] P. McGettrick, MIDIMatch: Musical pattern matching in real time, MSc Dissertation, York University, UK, 1997.

[16] G. Navarro, R. Baeza-Yates, Improving an algorithm for approximate string matching, Algorithmica 30 (4) (2001) 473–502.

[17] G. Navarro, M. Raffinot, Fast and flexible string matching by combining bit-parallelism and suffix automata, ACM J. Experimental Algorithmics (JEA) 5 (4) (2000).

[18] G. Navarro, M. Raffinot, Flexible Pattern Matching in Strings—Practical On-line Search Algorithms for Texts and Biological Sequences, Cambridge University Press, 2002.

[19] P. Roland, J. Ganascia, Musical pattern extraction and similarity assessment, in: E. Miranda (Ed.), Readings in Music and Artificial Intelligence, Harwood Academic Publishers, 2000, pp. 115–144.

[20] L.A. Smith, E.F. Chiu, B.L. Scott, A speech interface for building musical score collections, in: Proc. 5th ACM Conference on Digital Libraries, ACM Press, 2000, pp. 165–173.

[21] D. Sunday, A very fast substring searching algorithm, Comm. ACM 33 (8) (1990) 132–142.

[22] E. Ukkonen, Approximate string matching over suffix trees, in: Proc. 4th Ann. Symp. on Combinatorial Pattern Matching (CPM'93), 1993, pp. 228–242.

[23] S. Wu, U. Manber, Fast text searching allowing errors, Comm. ACM 35 (10) (1992) 83–91.