Efficient Algorithms for the δ-Approximate String Matching Problem in Musical Sequences

Domenico Cantone, Salvatore Cristofaro and Simone Faro

Dipartimento di Matematica e Informatica, Università di Catania, Italy

e-mail: {cantone, cristofaro, faro}@dmi.unict.it

Abstract. The δ-approximate string matching problem, recently introduced in connection with applications to music retrieval, is a generalization of the exact string matching problem for alphabets of integer numbers. In the δ-approximate variant, (exact) matching between any pair of symbols/integers a and b is replaced by the notion of δ-matching $=_\delta$, where $a =_\delta b$ if and only if $|a - b| \leq \delta$ for a given value of the approximation bound δ.
After surveying the state-of-the-art, we describe some new effective algorithms for the δ-matching problem, obtained by adapting existing string matching algorithms. The algorithms discussed in the paper are then compared with respect to a large set of experimental tests. From these, in particular it turns out that two of our newly proposed algorithms often achieve the best performances, especially in the case of large alphabets and short patterns, which typically occurs in practical situations in music retrieval.

Keywords: String algorithms, approximate string matching, musical information retrieval.

1 Introduction

Given a text T and a pattern P over some alphabet Σ, the *string matching problem* consists in finding *all* occurrences of the pattern P in the text T. It is a very extensively studied problem in computer science, mainly due to its direct applications to such diverse areas as text, image and signal processing, speech analysis and recognition, information retrieval, computational biology and chemistry, etc.

In the last few years also the approximate pattern matching problem has received much attention and algorithms which find all approximate repetitions of a given pattern in a sequence have been proposed, based on notions of approximation particularly useful in specific fields such as molecular biology [KMGL88, MJ93], musical applications [CIR98], or image processing [KPR00].

In this paper we focus on a variant of the approximate string matching problem which naturally arises in music information retrieval, namely the *δ-approximate string matching problem*.

Musical sequences can be schematically viewed as sequences of integer numbers, representing either the notes in the chromatic or diatonic notation (absolute pitch encoding), or the intervals, in number of semitones, between consecutive notes (pitch

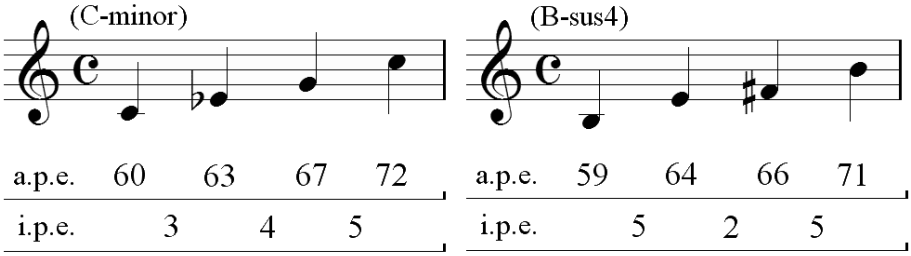

Figure 1: Representation of the C-minor and B-sus4 chords in the absolute pitch encoding (a.p.e.) and in the pitch interval encoding (i.p.e.)

interval encoding); see the examples in Figure 1. The second representation is generally of greater interest for applications in tonal music, since absolute pitch encoding disregards tonal qualities of pitches. Note durations and note accents can also be encoded in a numeric form, giving rise to more meaningful alphabets whose symbols can be really regarded as sets of parameters. This is the reason why alphabets used for music representation are generally quite large.

δ-approximate string matching algorithms are very effective to search for all similar but not necessarily identical occurrences of given (short) melodies in musical scores. We recall that in the δ-approximate matching problem two numeric strings of the same length match if corresponding integers differ by at most a fixed bound δ. For instance, the chords C-minor and B-sus4 match if a tolerance of $\delta = 1$ is allowed in the absolute pitch encoding (where C-minor= $(60, 63, 67, 72)$ and B-sus4= $(59, 64, 66, 71)$), whereas if we use the pitch interval encoding, a tolerance of $\delta = 2$ is needed to get a match (in this case we have C-minor= $(3, 4, 5)$ and B-sus4= $(5, 2, 5)$); see Figure 1. Notice that when $\delta = 0$, the δ-approximate string matching problem reduces to the exact string matching problem.

A stronger restriction can be introduced to δ-approximate matching by imposing a limit γ to the sum of the absolute differences between corresponding integers. This further restriction is generally referred to as (δ, γ)-approximate matching. However, in this paper we consider only the general case in which $\gamma = +\infty$.

A significant amount of research has been devoted to adapt solutions for exact string matching to δ-approximate matching (see for instance [CCI+99, CILP01, CIL+02]). In this respect, Boyer-Moore-type algorithms are of particular interest, since they are very fast. We recall that they are based on variations of the well-known ideas introduced in the Boyer-Moore algorithm [BM77], namely right-to-left scanning, bad-character and good-suffix heuristics. For instance, the Fast-Search and the Forward-Fast-Search algorithms [CF03a, CF03b] require that the bad-character heuristics is used only if the mismatching character is the last character of the pattern, otherwise the good-suffix heuristics is to be used.

The main results of this paper are adaptations of the Fast-Search and Forward-Fast-Search algorithms to δ-approximate matching. In addition, we propose adaptations of the Quick-Search and the Berry-Ravindran algorithms [Sun90, BR99], which are among the most efficient algorithms for exact string matching.

The paper is organized as follows. In Section 2 we introduce the basic notions and give a formal definition of the δ-approximate matching problem. In Section 3 we

survey some of the most efficient algorithms for computing δ-approximate repetitions in musical sequences. Then in Sections 4 and 5 we present new efficient variants, by suitably adapting known exact string matching algorithms. Experimental data obtained by running under various conditions the most efficient reviewed algorithms are presented and compared in Section 6. Finally, we draw our conclusions in Section 7.

2 Basic definitions and properties

Before entering into details, we need a bit of notations and terminology. A string P is represented as a finite array $P[0\,..\,m-1]$, with $m \geq 0$. In such a case we say that P has length m and write $\text{length}(P) = m$. In particular, for $m = 0$ we obtain the empty string, also denoted by ε. By $P[i]$ we denote the $(i+1)$-st character of P, for $0 \leq i < \text{length}(P)$. Likewise, by $P[i\,..\,j]$ we denote the substring of P contained between the $(i+1)$-st and the $(j+1)$-st characters of P, for $0 \leq i \leq j < \text{length}(P)$. Moreover, for any $i, j \in \mathbb{Z}$, we put

$$P[i\,..\,j] = \begin{cases} \varepsilon & \text{if } i > j \\ P[\max(i,0), \min(j, \text{length}(P)-1)] & \text{otherwise.} \end{cases}$$

For any two strings P and Q, we write $Q \sqsupset P$ to indicate that Q is a suffix of P, i.e., $Q = P[i\,..\,\text{length}(P)-1]$, for some $0 \leq i \leq \text{length}(P)$. Similarly, we write $Q \sqsubset P$ to indicate that Q is a prefix of P, i.e., $Q = P[0\,..\,i-1]$, for some $0 \leq i \leq \text{length}(P)$. In addition, we write $P.Q$ to denote the concatenation of P and Q and P^k to denote the concatenation of k copies of P. A prefix Q of P is a *period* of P if P is a prefix of Q^k, for a sufficiently large k. The shortest period of P is called *the period* of P.

Let Σ be an alphabet of integer numbers and let $\delta \geq 0$ be an integer. Two symbols a and b of Σ are said to be *δ-approximate* (or that a and b *δ-match*), in which case we write $a =_\delta b$, if $|a - b| \leq \delta$. Two strings P and Q over the alphabet Σ are said to be δ-approximate (or that P and Q δ-match), in which case we write $P \overset{\delta}{=} Q$, if

$$\text{length}(P) = \text{length}(Q), \quad \text{and } P[i] =_\delta Q[i], \text{ for } i = 0, ..., \text{length}(P) - 1\,.$$

Moreover, we write $Q \overset{\delta}{\sqsupset} P$, and say that Q is a *δ-suffix* of P, if $Q \overset{\delta}{=} P[i\,..\,\text{length}(P)-1]$, for some $0 \leq i \leq \text{length}(P)$. The following elementary property can be verified immediately.

Property 2.1 *Let P, Q, and R be strings over an alphabet Σ of integer numbers and let $\delta \geq 0$ be a given integer number. If $P \overset{\delta}{=} Q$ and $R \overset{\delta}{\sqsupset} Q$, then $R \overset{2\delta}{\sqsupset} P$.*

Let T be a text of length n and let P be a pattern of length m (over a given alphabet of integers). When the symbol $P[0]$ is aligned with the symbol $T[s]$ of the text, so that the symbol $P[i]$ is aligned with the symbol $T[s+i]$, for $i = 0, \ldots, m-1$, we say that the pattern P has *shift* s in T. In this case, the substring $T[s\,..\,s+m-1]$ is called the *current window* of the text. If $T[s\,..\,s+m-1] \overset{\delta}{=} P$, we say that the shift s is *δ-valid*. Then the problem of δ-approximate pattern matching consists in finding all δ-valid shifts of text T for a given pattern P.

3 Fast δ-Approximate Pattern Matching

The problem of δ-approximate matching in musical sequences has been formally defined in [CCI+99], where an algorithm based on the bitwise technique, the Shift-And algorithm, has been presented. The Shift-And algorithm uses a constant time state computation, for each character in the text, so that the overall time complexity of the searching phase is $\mathcal{O}(n)$.

Two years later, a number of efficient algorithms for the exact string matching problem have been adapted in [CILP01] to the δ-approximate matching problem, obtaining three algorithms based on occurrence heuristics (δ-Tuned-Boyer-Moore, δ-Skip-Search, and δ-Maximal-Shift) which are faster than the Shift-And algorithm.

Still later, other adaptations of exact string matching algorithms to the δ-approximate matching problem have been proposed in [CIL+02]. The resulting algorithms, based on the substring heuristics, are δ-BM1, δ-BM2, and δ-BM3.

Finally, a bit-parallel algorithm, δ-BNDM, which outperforms previous algorithms, has been recently presented in [CIPN03].

Next, we briefly review the most efficient algorithms for δ-approximate matching mentioned above.

3.1 δ-Boyer-Moore Algorithms

The Boyer-Moore algorithm [BM77] for exact pattern matching is the progenitor of several algorithmic variants which aim at computing efficiently shift increments which are close to optimal. Specifically, the Boyer-Moore algorithm checks whether a shift s of a text T is valid by scanning the pattern P from right to left. At the end of the matching phase, the Boyer-Moore algorithm computes the shift increment as the maximum value suggested by the *good-suffix rule* and the *bad-character rule*.

The Boyer-Moore bad-character rule can be easily adapted to δ-approximate matching. Suppose that the first δ-mismatch occurs at position i of the pattern, i.e. $P[i+1,...,m-1] \stackrel{\delta}{=} T[s+i+1,...,s+m-1]$ and $T[s+i] \neq_\delta P[i]$. Then the δ-bad-character rule suggests the shift increment $\delta\text{-}bc_P(T[s+i]) + i - m + 1$, where

$$\delta\text{-}bc_P(c) =_{\text{Def}} \min(\{0 \leq k < m \mid P[m-1-k] =_\delta c\} \cup \{m\}) \ ,$$

for $c \in \Sigma$.

The δ-Tuned-Boyer-Moore [CILP01] is an adaptation of Tuned-Boyer-Moore [HS91], which in turn is a very efficient simplification of the original Boyer-Moore algorithm although it runs in time $\mathcal{O}(nm)$ in the worst case. Specifically, each of its iterations can be divided into two phases: *last character localization* and *δ-matching phase*. The first phase searches for a δ-match of the character $P[m-1]$, by applying until needed rounds of three consecutive shifts based on the δ-bad-character rule, where the check $\delta\text{-}bc_P(T[s+m-1]) = 0$ is performed only at the end of each round. The subsequent matching phase then tries to δ-match the rest of the pattern $P[0\,..\,m-2]$ with the corresponding characters of the text, proceeding from right to left. At the end of the matching phase, the δ-Tuned-Boyer-Moore algorithm computes the shift advancement in such a way that character $T[s+m-1]$ is aligned with the rightmost position i on $P[0\,..\,m-2]$ such that $P[m-1] =_{2\delta} P[i]$, if present. If such position is not present, the shift is incremented by m.

The δ-Maximal-Shift algorithm, presented in [CILP01], is a modification of the Maximal-Shift algorithm [Sun90]. Rather than scanning the pattern from right to left, the δ-Maximal-Shift algorithm scans the pattern according to an ordering which guarantees larger shifts advancements. This is better formalized by means of a permutation $\pi : \{0, 1, ..., m\} \to \{0, 1, ..., m\}$ and a function $\delta\text{-}max : \{0, 1, ..., m\} \to \{0, 1, ..., m\}$ such that, for $0 \leq i < m$,

$$\delta\text{-}max(\pi(i)) = \min\{l \,|\, P[\pi(j)-l] =_{2\delta} P[\pi(j)] \text{ and } P[\pi(i)-l] \neq_{\delta} P[\pi(i)], \text{ for } 1 \leq j < i\}$$

and, for $0 \leq i < m-1$,

$$\delta\text{-}max(\pi(i)) \leq \delta\text{-}max(\pi(i+1))\,,$$

where P is a given pattern of length m. Furthermore, one sets $\pi(m) = m$ and $\delta\text{-}max(m)$ equal to the length of the period of P.

During the matching phase, characters are scanned in the pattern following the ordering $\pi(0), \pi(1), \ldots, \pi(m-1)$. Moreover, if the first δ-mismatch occurs while comparing characters $P[\pi(i)]$ and $T[s+\pi(i)]$, then the current shift s is incremented by $\max\{\delta\text{-}max(\pi(i)), \delta\text{-}bc(T[s+m])\}$. It turns out that the δ-Maximal-Shift algorithm has $\mathcal{O}(nm)$ time complexity.

3.2 δ-Reverse-Factor and δ-Alpha-Skip-Search Algorithms

Unlike the Boyer-Moore algorithm, the Reverse-Factor algorithm [CCG+94] and the Alpha-Skip-Search algorithm [CLP98] compute shifts which match prefixes of the pattern, rather than suffixes. These algorithms have a quadratic worst-case time complexity, but are very fast in practice (cf. [Lec00]). Moreover, it has been shown that on the average the Reverse-Factor algorithm inspects $\mathcal{O}(n \log(m)/m)$ text characters, reaching the best bound shown by Yao in 1979 (cf. [Yao79]).

Adaptations of Reverse-Factor and Alpha-Skip-Search algorithms are presented in [CIL+02] under the names δ-BM2 and δ-BM1, respectively.

The δ-BM1 algorithm, or δ-Alpha-Skip-Search algorithm, preliminary computes a δ-suffix trie $\mathcal{T}_x$ of all the factors of length $\ell = \lfloor \log_{|\Sigma|} m \rfloor$ occurring in the pattern P, where Σ is the alphabet. The leaves of the δ-suffix trie $\mathcal{T}_x$ represent all strings y such that $y \stackrel{\delta}{=} x$, for some factor x of P of length ℓ. For each leaf of $\mathcal{T}_x$, a bucket is maintained which stores all positions at which the factor associated to the leaf occurs in P. The searching phase of the δ-Alpha-Skip-Search algorithm consists then in looking for each shift position s into the bucket of the factor $T[s..s+\ell-1]$, if any, and in checking naively the corresponding windows of the text. A shift advancement of size $m - \ell + 1$ then takes place.

The δ-BM2 algorithm, or δ-Reverse-Factor algorithm, computes the smallest δ-suffix automaton of the reverse of the pattern P by simply minimizing the δ-suffix trie $\mathcal{T}_x$. In this way one obtains a deterministic finite automaton whose accepted language is the set of strings y such that $y \stackrel{\delta}{=} x$, for some factor x of P of length $\lfloor \log_{|\Sigma|} m \rfloor$. Then, much the same strategy of the Reverse-Factor algorithm can be applied to the case of δ-approximate matching.

3.3 δ-Hashing Algorithms

To describe the δ-BM3 algorithm, we need some further notation. Let P be a pattern over an alphabet Σ of integer numbers, let $k < \text{length}(P)$ be a fixed integer, and let $\delta \geq 0$ be a given approximation bound. We denote by $sub(P, k)$ the set of all substrings of P of length k and we define the following intervals:

$$\begin{aligned} \mathcal{I} &= [k \cdot \min \Sigma \,..\, k \cdot \max \Sigma] \\ \mathcal{I}_x &= [\max\{\text{hash}(x) - k\delta, k \cdot \min \Sigma\} \,..\, \min\{\text{hash}(x) + k\delta, k \cdot \max \Sigma\}] , \end{aligned}$$

for $x \in sub(P, k)$ and where hash(x) denotes the sum of the symbols of x. It can be easily shown that $\mathcal{I}_x \subseteq \mathcal{I}$, for each $x \in sub(P, k)$.

The δ-BM3 algorithm [CIL+02] begins by constructing the following hash table $\mathcal{H}$, indexed by the interval $\mathcal{I}$. For each $i \in \mathcal{I}$, the i-th bucket of the table $\mathcal{H}$ collects the positions of all subwords $x \in sub(P, k)$ such that $i \in \mathcal{I}_x$. Then, given a text T, for each shift position s the searching phase of the δ-BM3 algorithm consists in examining the subword $y = T[s+m-k \,..\, s+m-1]$. For each element j in the bucket at position $hash(y)$, the algorithm checks naively whether P occurs at position $s + m - k - j$ in T. It turns out that the choice of k influences the running-time of the algorithm. Generally, a value of $k = 2$ constitutes a good choice.

The δ-Skip-Search algorithm [CILP01] is an adaptation of the Skip-Search algorithm [CLP98] to δ-approximate matching. However, it can also be seen as a variant of the δ-BM3 algorithm, with $k = 1$.

Both δ-Skip-Search and δ-BM3 algorithms are fast in practice although their worst-case time complexity is $\mathcal{O}(nm)$.

3.4 The δ-BNDM Algorithm

The Backward Nondeterministic DAWG Matching algorithm (BNDM for short) for exact string matching has been presented in [NR98] as a combination of the bit-parallel algorithm Shift-Or [BYG89] and the BDM algorithm [CCG+94] based on suffix automata. The aim of the BNDM algorithm is to combine the property of skipping characters (as in the BDM algorithm) with that of simulating nondeterministic automata (as the Shift-Or algorithm). It turns out that the BNDM algorithm obtains better results in terms of running time than the Shift-Or and the BDM algorithms. Its complexity is $\mathcal{O}(nm)$ in the worst case.

An adaptation of the BNDM algorithm to δ-approximate matching, the δ-BNDM algorithm, has been presented in [CIPN03]. The δ-BNDM algorithms is very simple and efficient, especially in the case of long patterns, and is considered a stronger choice for the δ-approximate matching problem.

4 δ-Fast-Search algorithms

The Fast-Search [CF03a] and the Forward-Fast-Search [CF03b] algorithms are very recent members of the large family of Boyer-Moore type string matching algorithms. Their searching strategy is based on an efficient mixing of the bad-character and good-suffix rules, as given in the original Boyer-Moore algorithm. Recent experimental results [CF03b] conducted over an extensive family of string matching algorithms show that the Fast-Search algorithms obtain, in most cases, the best results in terms

of running times and number of text character inspections.

After reviewing the main features of the Fast-Search algorithms, we shall show how they can be adapted to δ-approximate matching problem.

4.1 Fast-Search and Forward-Fast-Search algorithms

The Fast-Search algorithm is a very simple, yet efficient, variant of the Boyer-Moore algorithm. Again, let P be a pattern of length m and let T be a text of length n over a finite alphabet Σ. The Fast-Search algorithm computes its shift increments by applying the bad-character rule if and only if a mismatch occurs during the first character comparison, namely, while comparing characters $P[m-1]$ and $T[s+m-1]$, where s is the current shift. Otherwise it uses the good-suffix rule.

Specifically, if the first mismatch occurs at position $i < m-1$ of the pattern P, the good-suffix rule suggests to align the substring $T[s+i+1\,..\,s+m-1] = P[i+1\,..\,m-1]$ with its rightmost occurrence in P preceded by a character different from $P[i]$. If such an occurrence does not exist, the good-suffix rule suggests a shift increment which allows to match the longest suffix of $T[s+i+1\,..\,s+m-1]$ with a prefix of P.

More formally, if the first mismatch occurs at position i of the pattern P, the good-suffix rule states that the shift can be safely incremented by $gs_P(i+1)$ positions, where

$$gs_P(j) =_{\text{Def}} \min\{0 < k \leq m \quad | \quad P[j-k\,..\,m-k-1] \sqsupset P \text{ and } (k \leq j-1 \rightarrow P[j-1] \neq P[j-1-k])\} \ ,$$

for $j = 0, 1, \ldots, m$. (The situation in which an occurrence of the pattern P is found can be regarded as a mismatch at position -1.)

A more effective implementation of the Fast-Search algorithm is obtained along the same lines of the Tuned-Boyer-Moore algorithm: the bad-character rule can be iterated until the last character $P[m-1]$ of the pattern is matched correctly against the text. At this point it is known that $T[s+m-1] = P[m-1]$, so that the subsequent matching phase can start with the $(m-2)$-*nd* character of the pattern. At the end of the matching phase the algorithm uses the good-suffix rule for shifting. Moreover the Fast-Search algorithm benefits from the introduction of an external sentinel, which allows to compute correctly the last shifts with no extra checks.

The Forward-Fast-Search algorithm maintains the same structure of the Fast-Search algorithm, but it is based upon a modified version of the good-suffix rule, called *forward good-suffix* rule, which uses a look-ahead character to determine larger shift advancements. Thus, if the first mismatch occurs at position $i < m-1$ of the pattern P, the forward good-suffix rule suggests to align the substring $T[s+i+1\,..\,s+m]$ with its rightmost occurrence in P preceded by a character different from $P[i]$. If such an occurrence does not exist, the forward good-suffix rule proposes a shift increment which allows to match the longest suffix of $T[s+i+1\,..\,s+m]$ with a prefix of P. This corresponds to advance the shift s by $\overrightarrow{gs}_P(i+1, T[s+m])$ positions, where

$$\overrightarrow{gs}_P(j,c) =_{\text{Def}} \min(\{0 < k \leq m \quad | \quad P[j-k\,..\,m-k-1] \sqsupset P \text{ and } (k \leq j-1 \rightarrow P[j-1] \neq P[j-1-k]) \text{ and } P[m-k] = c\} \cup \{m+1\}) \ ,$$

for $j = 0, 1, \ldots, m$ and $c \in \Sigma$.

The good-suffix rule and the forward good-suffix rule require tables of size m and $m \cdot |\Sigma|$, respectively. These can be constructed in time $\mathcal{O}(m)$ and $\mathcal{O}(m \cdot \max(m, |\Sigma|))$, respectively.

4.2 Adaptations to δ-approximate matching

In this section we show how to adapt the Fast-Search and Forward-Fast-Search algorithms to δ-approximate matching.

A modification of the bad-character rule to δ-approximate matching has been already presented in Section 3.1. Now we show how the good-suffix rule and the forward good-suffix rule can also be adapted to match δ-approximate repetitions of suffixes of the pattern.

Let us suppose that while comparing the pattern P with the window $T[s \,..\, s + m - 1]$, proceeding from right to left, the first δ-mismatch occurs at position i, i.e. $P[i] \neq_\delta T[s + i]$ and $P[i + 1 \,..\, m - 1] \stackrel{\delta}{=} T[s + i + 1 \,..\, s + m - 1]$ (if we have a δ-match, then $i = 0$ and the condition $P[i] \neq_\delta T[s + i]$ should not be considered). Let $0 < k \leq m$ be such that $s + k + m - 1 \leq n$ and $P[i + 1 - k \,..\, m - 1 - k] \stackrel{2\delta}{\not\sqsupset} P$, where $\stackrel{\delta}{\sqsupset}$ is the δ-suffix relation defined in Section 2. Then the shift $s + k$ is not δ-valid. Indeed, if $s + k$ were δ-valid, we would have $P \stackrel{\delta}{=} T[s + k \,..\, s + k + m - 1]$, so that $P[i+1-k \,..\, m-1-k] \stackrel{\delta}{\sqsupset} T[s+i+1 \,..\, s+m-1]$. Therefore, by Property 2.1, we would get $P[i+1-k \,..\, m-1-k] \stackrel{2\delta}{\sqsupset} P[i+1 \,..\, m-1]$, which yields $P[i+1-k \,..\, m-1-k] \stackrel{2\delta}{\sqsupset} P$, a contradiction. It is also easy to verify that if an integer k satisfies both $0 < k \leq i$ and $P[i] = P[i - k]$, then again the shift $s + k$ is not δ-valid. The above considerations allow us to state the following *δ-good-suffix rule*: if the first δ-mismatch occurs at position i of the pattern P, then the shift can be safely incremented by $\delta\text{-}gs_P(i + 1)$ positions, where

$$\delta\text{-}gs_P(j) =_{\text{Def}} \min\{0 < k \leq m \quad | \quad P[j - k \,..\, m - k - 1] \stackrel{2\delta}{\sqsupset} P[j \,...\, m - 1] \text{ and } (k \leq j - 1 \rightarrow P[j - 1] \neq P[j - 1 - k])\} \ ,$$

for $j = 0, 1, \ldots, m$.

Much in the same way, one can verify the correctness of the following *δ-forward good-suffix rule*: if the first δ-mismatch occurs at position i of the pattern P, then the shift can be safely incremented by $\delta\text{-}\overrightarrow{gs}_P(i + 1, T[s + m])$ positions, where

$$\delta\text{-}\overrightarrow{gs}_P(j, c) =_{\text{Def}} \min(\{0 < k \leq m \quad | \quad P[j - k \,..\, m - k - 1] \stackrel{2\delta}{\sqsupset} P[j \,...\, m - 1] \text{ and } (k \leq j - 1 \rightarrow P[j - 1] \neq P[j - 1 - k]) \text{ and } P[m - k] =_\delta c\} \cup \{m + 1\}) \ ,$$

for $j = 0, 1, \ldots, m$ and $c \in \Sigma$.

The δ-good-suffix rule and the δ-forward good-suffix rule require tables of size m and $(m \cdot |\Sigma|)$, respectively. These can be easily constructed in time $\mathcal{O}(m)$ and $\mathcal{O}(\delta \cdot m \cdot \max(m, |\Sigma|))$, respectively.

The δ-Fast-Search and δ-Forward-Fast-Search algorithms can be implemented much along the same lines of the δ-Tuned-Boyer-Moore algorithm.

```
δ-Fast-Search (P, T)
    n = length(T)
    m = length(P)
    T' = T.P[m − 1]^{m+1}
    δ-bc = precompute-δ-bad-character(P)
    δ-gs = precompute-δ-good-suffix(P)
    s = 0
    while δ-bc[T'[s + m − 1]] > 0 do
        s = s + δ-bc[T'[s + m − 1]]
    while s ≤ n − m do
        j = m − 2
        while j ≥ 0 and P[j] =_δ T'[s + j]
            do  j = j − 1
        if j < 0 then
            print(s)
        s = s + δ-gs[j + 1]
        while δ-bc[T'[s + m − 1]] > 0 do
            s = s + δ-bc[T'[s + m − 1]]
```

```
δ-Forward-Fast-Search (P, T)
    n = length(T)
    m = length(P)
    T' = T.P[m − 1]^{m+1}
    δ-bc = precompute-δ-bad-character(P)
    δ-→gs = precompute-δ-forward-good-suffix(P)
    s = 0
    while δ-bc[T'[s + m − 1]] > 0 do
        s = s + δ-bc[T'[s + m − 1]]
    while s ≤ n − m do
        j = m − 2
        while j ≥ 0 and P[j] =_δ T'[s + j]
            do  j = j − 1
        if j < 0 then
            print(s)
        s = s + δ-→gs[j + 1, T[s + m]]
        while δ-bc[T'[s + m − 1]] > 0 do
            s = s + δ-bc[T'[s + m − 1]]
```

Figure 2: δ-Fast-Search and δ-Forward-Fast-Search algorithms

Each iteration of both algorithms can be divided into two phases. In the first phase, called *character localization* phase, the δ-bad-character rule is iterated until the last character $P[m-1]$ of the pattern is δ-matched correctly against the text. More precisely, starting from a shift position s, if we denote by j_i the total shift advancement after the i-th iteration of the δ-bad-character rule, then we have the following recurrence:

$$j_i = j_{i-1} + \delta\text{-}bc_P(T[s + j_{i-1} + m - 1]) \ .$$

Therefore, the δ-bad-character rule is applied k times in a row, where $k = \min\{i \mid T[s+ j_i + m - 1] =_\delta P[m-1]\}$, with an overall shift advancement of j_k.

At this point we have that $T[s + j_k + m - 1] =_\delta P[m-1]$, so that the subsequent *δ-matching* phase can test for a δ-occurrence of the pattern by comparing only the remaining $(m-1)$ characters of the pattern. At the end of the δ-matching phase, the δ-good-suffix or the δ-forward-good-suffix rule is applied for computing the next shift.

In order to compute correctly the last shift with no extra checks, it is convenient to add $m+1$ copies of the character $P[m-1]$ at the end of the text T, obtaining the new text $T' = T.P[m-1]^{m+1}$. Plainly, all the δ-valid shifts of P in T are exactly the δ-valid shifts s of P in T' such that $s \leq n-m$, where, as usual, n and m denote respectively the lengths of T and P. The codes of the δ-Fast-Search and δ-Forward-Fast-Search algorithms are presented in Figure 2.

5 Other interesting efficient variants

In this section we present adaptations to δ-approximate matching of two efficient exact string matching algorithms based on the bad-character rule, i.e. the Quick-Search algorithm and the Berry-Ravindran algorithm.

The Quick-Search algorithm, presented in [Sun90], uses a simple modification of the original heuristics of the Boyer-Moore algorithm. Specifically, it is based on the following observation: when a mismatch character is encountered, the pattern is always shifted to the right by at least one character, but never by more than m characters. Thus, the character $T[s+m]$ is always involved in testing for the next alignment. So, one can apply the bad-character rule to $T[s+m]$, rather than to the mismatching character, obtaining larger shift advancements. Moreover the good-suffix rule of the original Boyer-Moore algorithm is not used at all.

Extending this idea to δ-approximate matching we obtain the δ-Quick-Search algorithm which, after each δ-matching phase, advances the shift by $\delta\text{-}qbc_P(T[s+m])$ positions, where

$$\delta\text{-}qbc_P(c) =_{\text{Def}} \min(\{0 < k \leq m \mid P[m-k] =_\delta c\} \cup \{m+1\}) \ .$$

The function $\delta\text{-}qbc_P$ can be precomputed in $\mathcal{O}(m \cdot \delta + |\Sigma|)$-time and $\mathcal{O}(|\Sigma|)$-space complexity.

The δ-Berry-Ravindran algorithm is a modification of the Berry-Ravindran algorithm [BR99]. It extends the δ-Quick-Search algorithm in that its bad-character rule uses the two characters $T[s+m]$ and $T[s+m+1]$ rather than just the last character $T[s+m]$ of the window. Thus, at the end of each matching phase with shift s, the δ-Berry-Ravindran algorithm advances the pattern in such a way that the substring of the text $T[s+m\,..\,s+m+1]$ is aligned with the rightmost δ-occurrence in P of a substring c_1c_2 such that $T[s+m\,...\,s+m+1] \overset{\delta}{=} c_1c_2$.

The precomputation of the table used by this version of δ-bad-character rule requires $\mathcal{O}(m \cdot \delta^2 + |\Sigma|^2)$-time and $\mathcal{O}(|\Sigma|^2)$-space complexity.

Experimental results confirm the good practical behavior of the Quick-Search and Berry-Ravindran algorithms even in the case of their δ-variants (see next section).

6 Experimental Results

In this section we report experimental data related to the most efficient δ-approximate string matching algorithms described above, namely δ-Tuned-Boyer-Moore (δ-TBM), δ-Quick-Search (δ-QS), δ-Berry-Ravindran (δ-BR), δ-BNDM (δ-BNDM), δ-Fast-Search (δ-FS), and δ-Forward-Fast-Search (δ-FFS).

We have chosen to compare them in terms of their running time. All algorithms have been implemented in the **C** programming language and were used to search for the same patterns in large fixed text sequences on a PC with a Pentium IV processor at 2.6GHz. In particular, they have been tested on three Randσ problems, for $\sigma = 30, 60, 120$, and on a real music sequence with patterns of length $m = 2, 4, 6, 8, 10, 15, 20, 25$, and 30.

Each Randσ problem consists in searching a set of 300 random patterns of a given length in a 20Mb random text sequence over a common alphabet of size σ.

The tests on the real music text buffer have been performed on a 9.3Mb file obtained by combining a set of classical pieces, in MIDI format, by J.S. Bach. The resulting text buffer has been translated in the pitch interval encoding with an alphabet of 55 symbols. For each pattern length m, we have randomly selected 200 substrings of length m in the file which subsequently have been searched for in the same file.

For both Randσ problems and real music problems, the value of the bound δ has been set to 1, 2, and 4.

In the following tables running times have been expressed in milliseconds and, for each length of the pattern, the best results have been bold-faced.

From the experimental results, it turns out that the δ-BNDM algorithm is a very good choice for the δ-approximate matching problem, especially when the pattern is long or the size of the alphabet is small.

However, the δ-Fast-Search algorithms compares well with the δ-BNDM algorithm and outperforms it in the case of quite short patterns and large alphabets, which occurs most frequently in real musical information retrieval problems.

Observe also that the δ-Tuned-Boyer-Moore algorithm and the δ-Quick-Search algorithm obtain good results in most cases and, additionally, the δ-Quick-Search algorithm reaches competitive results in the case of long patterns and large alphabets.

$\sigma = 30, \delta = 1$	2	4	6	8	10	15	20	25	30
δ-QS	93.08	68.88	59.19	54.48	51.71	49.12	48.23	48.16	47.85
δ-TBM	75.83	55.44	49.84	47.82	47.42	46.63	46.02	45.57	46.02
δ-BR	140.67	102.82	84.48	73.38	65.99	55.33	51.11	48.68	47.44
δ-FFS	74.67	54.84	49.45	**47.42**	**46.57**	**45.85**	**45.39**	45.73	45.10
δ-FS	**74.35**	**54.11**	**49.21**	47.63	46.92	46.00	45.56	45.54	45.64
δ-BNDM	125.89	90.41	73.61	62.92	55.80	48.03	46.10	**45.29**	**44.46**

Rand30 problem with $\delta = 1$

$\sigma = 30, \delta = 2$	2	4	6	8	10	15	20	25	30
δ-QS	118.91	87.03	73.50	67.25	63.04	58.90	57.20	56.82	56.26
δ-TBM	94.59	69.52	59.58	56.80	54.40	53.07	52.49	52.40	52.00
δ-BR	171.74	125.04	102.18	87.26	78.73	65.21	57.87	53.57	51.79
δ-FFS	**93.44**	**67.40**	**58.26**	**54.41**	**52.21**	**49.68**	48.36	48.10	47.92
δ-FS	93.80	68.19	58.54	55.76	52.95	51.40	51.29	50.05	49.91
δ-BNDM	158.89	107.54	80.57	66.42	58.79	50.21	**47.57**	**46.78**	**45.26**

Rand30 problem with $\delta = 2$

$\sigma = 30, \delta = 4$	2	4	6	8	10	15	20	25	30
δ-QS	166.00	132.47	112.36	105.59	101.12	98.74	97.91	98.79	97.91
δ-TBM	**124.15**	103.95	92.90	91.22	88.08	88.79	87.16	87.17	86.58
δ-BR	229.80	178.12	145.58	126.55	114.72	96.98	88.09	83.17	79.27
δ-FFS	127.36	**101.16**	**85.88**	**80.17**	75.31	70.70	67.14	65.06	63.38
δ-FS	131.92	105.31	92.44	88.90	84.92	84.04	80.76	80.22	78.55
δ-BNDM	215.78	134.42	98.63	82.78	**72.84**	**58.07**	**51.43**	**48.19**	**47.18**

Rand30 problem with $\delta = 4$

$\sigma = 60, \delta = 1$	2	4	6	8	10	15	20	25	30
δ-QS	74.92	57.49	51.07	48.74	47.46	46.02	45.45	45.35	45.27
δ-TBM	62.48	49.22	46.21	45.96	46.53	45.54	44.85	44.95	44.49
δ-BR	122.15	89.97	74.42	65.24	58.70	50.39	48.54	46.46	46.25
δ-FFS	62.63	49.08	46.71	45.39	**45.16**	45.14	**44.14**	44.71	44.17
δ-FS	**62.01**	**48.71**	**45.60**	**45.17**	45.40	**44.99**	44.49	**44.35**	**44.10**
δ-BNDM	102.68	72.16	62.16	56.48	52.32	47.60	45.79	44.60	44.22

Rand60 problem with $\delta = 1$

$\sigma = 60, \delta = 2$	2	4	6	8	10	15	20	25	30
δ-QS	86.46	64.67	55.84	52.05	50.04	47.76	47.03	46.48	46.11
δ-TBM	71.12	52.55	48.45	46.71	46.81	46.10	45.41	45.50	45.56
δ-BR	134.57	98.74	81.02	70.64	63.66	53.73	50.15	48.07	46.85
δ-FFS	**70.06**	52.58	48.42	**46.31**	**45.85**	45.60	45.40	**44.77**	45.03
δ-FS	70.21	**51.79**	**47.32**	46.36	46.02	**45.17**	**44.97**	44.91	**44.43**
δ-BNDM	117.93	84.61	70.87	62.07	55.01	47.77	45.73	44.98	44.77

Rand60 problem with $\delta = 2$

$\sigma = 60, \delta = 4$	2	4	6	8	10	15	20	25	30
δ-QS	112.38	82.29	70.06	62.81	59.55	55.80	54.66	53.67	53.60
δ-TBM	89.94	65.92	56.49	53.92	52.14	50.57	49.99	50.34	49.69
δ-BR	164.51	120.44	98.35	84.06	75.90	62.74	56.23	52.55	50.81
δ-FFS	88.41	**63.94**	**54.83**	**52.07**	**50.56**	**48.34**	47.63	47.05	46.61
δ-FS	**88.33**	64.39	56.29	52.89	51.29	49.69	48.79	48.80	48.47
δ-BNDM	150.64	103.41	78.90	65.46	56.91	49.35	**46.95**	**46.40**	**45.50**

Rand60 problem with $\delta = 4$

$\sigma = 120, \delta = 1$	2	4	6	8	10	15	20	25	30
δ-QS	67.32	52.41	48.78	46.86	46.42	45.28	44.31	44.61	43.92
δ-TBM	57.44	47.79	46.17	45.60	45.85	45.21	45.72	44.51	44.28
δ-BR	114.24	85.12	70.86	62.88	56.61	49.86	46.96	47.11	46.31
δ-FFS	57.62	**47.74**	**45.39**	45.16	45.67	**44.56**	45.06	44.32	**43.89**
δ-FS	**57.15**	47.86	45.96	**44.82**	**45.06**	44.78	**43.20**	**44.13**	43.95
δ-BNDM	91.98	62.92	53.25	49.97	48.40	45.89	46.07	44.85	44.68

Rand120 problem with $\delta = 1$

$\sigma = 120, \delta = 2$	2	4	6	8	10	15	20	25	30
δ-QS	72.31	55.66	50.43	48.24	47.56	46.54	45.39	43.60	44.89
δ-TBM	60.73	48.71	46.57	45.60	45.82	44.80	45.39	46.17	44.93
δ-BR	119.57	89.09	73.42	64.91	58.82	50.80	47.82	46.08	45.98
δ-FFS	60.76	48.53	46.17	45.57	45.65	45.46	**44.60**	45.71	44.71
δ-FS	**60.30**	**48.25**	**45.89**	**45.06**	**44.91**	**44.69**	44.86	**42.88**	**43.89**
δ-BNDM	99.36	69.31	58.94	54.62	51.12	47.22	45.74	46.49	44.49

Rand120 problem with $\delta = 2$

$\sigma = 120, \delta = 4$	2	4	6	8	10	15	20	25	30
δ-QS	82.92	62.78	54.59	50.91	49.22	47.32	46.66	46.28	46.06
δ-TBM	69.13	51.62	48.01	47.07	46.36	45.63	45.76	45.30	45.28
δ-BR	132.63	97.17	80.17	69.71	62.69	53.69	49.81	48.12	47.18
δ-FFS	67.86	51.45	47.40	46.29	46.53	45.64	45.14	44.88	44.67
δ-FS	**67.61**	**50.91**	**47.32**	**45.92**	**45.60**	**45.43**	**44.99**	44.95	44.78
δ-BNDM	114.37	82.24	69.07	60.88	54.61	47.62	45.96	**44.79**	**44.42**

Rand120 problem with $\delta = 4$

$\sigma = 55, \delta = 1$	2	4	6	8	10	15	20	25	30
δ-QS	11.62	8.30	7.54	7.17	6.66	5.68	5.95	5.59	5.36
δ-TBM	9.40	7.33	6.47	5.82	5.62	5.51	5.29	5.51	5.83
δ-BR	16.16	12.56	10.10	8.79	7.64	6.97	6.11	5.87	5.44
δ-FFS	**8.76**	**6.58**	6.44	6.22	5.61	5.29	5.26	5.19	5.18
δ-FS	9.16	6.72	**5.79**	**5.51**	**5.59**	**5.06**	**5.19**	**5.11**	5.26
δ-BNDM	15.17	10.33	8.08	6.97	6.54	6.08	5.47	**5.11**	**5.01**

Results on the Real Music problem with $\delta = 1$

$\sigma = 55, \delta = 2$	2	4	6	8	10	15	20	25	30
δ-QS	14.53	11.41	9.79	9.51	8.63	8.03	7.61	7.68	7.20
δ-TBM	11.55	9.39	8.42	8.39	7.86	7.17	7.28	6.78	6.93
δ-BR	20.36	16.21	13.20	12.26	10.68	9.00	8.39	7.42	7.14
δ-FFS	11.35	**8.78**	**7.21**	**6.64**	6.73	**5.34**	5.43	6.10	5.89
δ-FS	**11.23**	8.97	7.78	7.74	7.78	6.88	6.60	6.38	6.34
δ-BNDM	18.36	12.85	9.72	8.06	**6.71**	6.19	**5.09**	**5.24**	**5.42**

Results on the Real Music problem with $\delta = 2$

$\sigma = 55, \delta = 4$	2	4	6	8	10	15	20	25	30
δ-QS	17.64	16.70	15.79	14.99	14.63	13.69	12.90	13.39	12.83
δ-TBM	15.18	15.19	14.30	13.80	13.02	12.54	11.76	11.92	11.54
δ-BR	26.12	23.17	21.36	19.73	18.33	15.60	14.12	14.22	13.14
δ-FFS	**14.02**	**13.28**	**12.23**	**11.33**	10.32	9.84	9.57	9.18	9.26
δ-FS	14.52	14.20	13.37	12.43	12.18	11.46	10.42	10.39	9.83
δ-BNDM	21.92	17.03	14.60	11.86	**9.80**	**7.79**	**6.75**	**5.85**	**5.60**

Results on the Real Music problem with $\delta = 4$

7 Conclusion

As reported in [CIL+02], typical problems arising in musical analysis and musical information retrieval generally use representations of musical scores requiring large alphabets. In such problems the length of the pattern is generally short (10-20 notes): thus the need of approximate searching algorithms that perform well for small patterns and large alphabets.

In this paper we have focused our attention on δ-approximate string matching algorithms, which are very effective in searching for all similar but not necessarily identical occurrences of given melodies in musical scores. In particular we have presented two new efficient algorithms, δ-Fast-Search and δ-Forward-Fast-Search, which outperform known algorithms in the case of small patterns and large alphabets.

References

[BM77] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.

[BR99] T. Berry and S. Ravindran. A fast string matching algorithm and experimental results. In J. Holub and M. Šimánek, editors, *Proceedings of the Prague Stringology Club Workshop '99*, pages 16–28, Czech Technical University, Prague, Czech Republic, 1999. Collaborative Report DC–99–05.

[BYG89] R. A. Baeza-Yates and G. H. Gonnet. A new approach to text searching. In N. J. Belkin and C. J. van Rijsbergen, editors, *Proceedings of the 12th International Conference on Research and Development in Information Retrieval*, pages 168–175, Cambridge, MA, 1989. ACM Press.

[CCG+94] M. Crochemore, A. Czumaj, L. Gąsieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter. Speeding up two string matching algorithms. *Algorithmica*, 12(4/5):247–267, 1994.

[CCI+99] E. Cambouropoulos, M. Crochemore, C. S. Iliopoulos, L. Mouchard, and Y. J. Pinzon. Algorithms for computing approximate repetitions in musical sequences. In R. Raman and J. Simpson, editors, *Proceedings of the 10th Australasian Workshop On Combinatorial Algorithms*, pages 129–144, Pert, WA, Australia, 1999.

[CF03a] D. Cantone and S. Faro. Fast-Search: A new efficient variant of the Boyer-Moore string matching algorithm. In K. Jansen, M. Margraf, M. Mastrolli, and J.D.P. Rolim, editors, *Proceedings of the 9th Workshop on Experimental Algorithms (WEA 2003)*, volume 2647 of *Lecture Notes in Computer Science*, pages 47–58. Springer-Verlag, Berlin, 2003.

[CF03b] D. Cantone and S. Faro. Forward-Fast-Search: Another fast variant of the Boyer-Moore string matching algorithm. In M. Šimánek, editor, *Proceedings of the Prague Stringology Conference '03*, pages 10–24, Czech Technical University, Prague, Czech Republic, 2003.

[CIL+02] M. Crochemore, C. S. Iliopoulos, T. Lecroq, W. Plandowski, and W. Rytter. Three heuristics for δ-matching: δ-BM algorithms. In A. Apostolico and M. Takeda, editors, *Proceedings of the 13th Annual Symposium on Combinatorial Pattern Matching*, number 2373 in Lecture Notes in Computer Science, pages 178–189, Fukuoka, Japan, 2002. Springer-Verlag, Berlin.

[CILP01] M. Crochemore, C. S. Iliopoulos, T. Lecroq, and Y. J. Pinzon. Approximate string matching in musical sequences. In M. Balík and M. Šimánek, editors, *Proceedings of the Prague Stringology Conference '01*, pages 26–36, Prague, Czech Republic, 2001. Annual Report DC–2001–06.

[CIPN03] M. Crochemore, C. S. Iliopoulos, Y. J. Pinzon, and G. Navarro. A bit-parallel suffix automaton approach for (δ, γ)-matching in music retrieval. In Edleno S. De Moura and A. L. Oliveira, editors, *Proc. of the 10th International Symposium on String Processing and Information Retrieval (SPIRE'03)*, number 2857 in lncs, pages 211–223. Svb, 2003.

[CIR98] T. Crawford, C. Iliopoulos, and R. Raman. String matching techniques for musical similarity and melodic recognition. *Computing in Musicology*, 11:71–100, 1998.

[CLP98] C. Charras, T. Lecroq, and J.D. Pehoushek. A very fast string matching algorithm for small alphabets and long patterns. In M. Farach-Colton, editor, *Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching*, number 1448 in Lecture Notes in Computer Science, pages 55–64. Springer-Verlag, Berlin, 1998.

[HS91] A. Hume and D. M. Sunday. Fast string searching. *Softw. Pract. Exp.*, 21(11):1221–1248, 1991.

[KMGL88] S. Karlin, M. Morris, G. Ghandour, and M. Y. Leung. Efficient algorithms for molecular sequence analysis. *Proceedings of the National Academy of Science*, 85:841–845, 1988.

[KPR00] J. Karhumäki, W. Plandowski, and W. Rytter. Pattern-matching problems for two-dimensional images described by finite automata. *Nordic J. Comput.*, 7(1):1–13, 2000.

[Lec00] T. Lecroq. New experimental results on exact string-matching. Rapport LIFAR 2000.03, Université de Rouen, France, 2000.

[MJ93] A. Milosavljevic and J. Jurka. Discovering simple DNA sequences by the algorithmic significance method. *Comp. Appl. BioSci.*, 9(4):407–411, 1993.

[NR98] G. Navarro and M. Raffinot. A bit-parallel approach to suffix automata: Fast extended string matching. Technical Report TR/DC–98–1, Department of Computer Science, University of Chile, 1998.

[Sun90] D. M. Sunday. A very fast substring search algorithm. *Commun. ACM*, 33(8):132–142, 1990.

[Yao79] A. C. Yao. The complexity of pattern matching for a random string. *SIAM J. Comput.*, 8(3):368–387, 1979.