open science

# Algorithms for computing approximate repetitions in musical sequences 

Emilios Cambouropoulos, Maxime Crochemore, Costas S. Iliopoulos, Laurent<br>Mouchard, Yoan J. Pinzón Ardila

## - To cite this version:

Emilios Cambouropoulos, Maxime Crochemore, Costas S. Iliopoulos, Laurent Mouchard, Yoan J. Pinzón Ardila. Algorithms for computing approximate repetitions in musical sequences. Australasian Workshop On Combinatorial Algorithms, 1999, Australia. pp.129-144. hal-00452240

HAL Id: hal-00452240

## https://hal.science/hal-00452240

Submitted on 20 Mar 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Algorithms for Computing Approximate Repetitions in Musical Sequences 

Emilios Cambouropoulos ${ }^{1 \star}$, Maxime Crochemore ${ }^{2 \star \star}$, Costas S. Iliopoulos ${ }^{3 \star \star \star}$, Laurent Mouchard ${ }^{4} \dagger$, and Yoan J. Pinzon ${ }^{3}$<br>${ }^{1}$ Austrian Research Institute for Artificial Intelligence, Schottengasse 3, 1010 Wien, Austria<br>emilios@ai.univie.ac.at<br>www.ai.univie.ac.at/~emilios<br>${ }^{2}$ Institut Gaspard Monge, Université de Marne-la-Vallée, 77454 Marne-la-Vallée CEDEX 2, France. mac@univ-mlv.fr<br>www-igm.univ-mlv.fr/~mac<br>${ }^{3}$ Dept. Computer Science, King's College London, London WC2R 2LS, England, and School of Computing, Curtin University of Technology, GPO Box 1987 U, WA. \{csi, pinzon\}@dcs.kcl.ac.uk,<br>www.dcs.kcl.ac.uk/staff/csi, www.dcs.kcl.ac.uk/pg/pinzon<br>${ }^{4}$ LIFAR - ABISS, Université de Rouen, 76821 Mont Saint Aignan, France. and School of Computing, Curtin University of Technology, GPO Box 1987 U, WA.<br>lm@dir.univ-rouen.fr<br>www.dir.univ-rouen.fr/~lm


#### Abstract

Here we introduce two new notions of approximate matching with application in computer assisted music analysis. We present algorithms for each notion of approximation: for approximate string matching and for computing approximate squares.


Keywords: String algorithms, approximate string matching, dynamic programming, computer-assisted music analysis.

## 1 Introduction

This paper focuses on a set of string pattern-matching problems that arise in musical analysis, and especially in musical information 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

[^0]pitch intervals as number of semitones). Approximate repetitions in one or more musical works play a crucial role in discovering similarities between different musical entities and may be used for establishing "characteristic signatures" (see [6]). Such algorithms can be particularly useful for melody identification and musical retrieval.

Both exact and approximate matching techniques have been used for a variety of musical applications (see overviews in McGettrick [23] ; Crawford et al [6]; Rolland et al [28]; Cambouropoulos et al [4]). The specific problem studied in this paper is pattern-matching for numeric strings where a certain tolerance is allowed during the matching procedure. This type of pattern-matching has been considered necessary for various musical applications and has been used by some researchers (see, for instance, Cope [5]). A number of efficient algorithms will be presented in this paper that tackle various aspects of this problem.

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 a 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 use what will be referred to as $\delta$-approximate matching (and $\gamma$-approximate matching). In $\delta$-approximate matching, equal-length patterns consisting of integers match if each corresponding integer differs by not more than $\delta$ - 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 ( $\gamma$-approximate matching is described in the next section). Two simple musical examples that illustrate the usefulness of the proposed pattern-matching techniques are presented in Appendices I and II.

Exact repetitions have been studied extensively. The repetitions can be either concatenated with the original substring or they may overlap or they may not. Algorithms for finding non-overlapping repetitions in a given string can be found in $[1,8,15,21,18,26]$ and algorithms for computing overlapping repetitions can be found in $[3,13,14,25]$. A natural extension of the repetitions problem is to allow the presence of errors; that is, the identification of substrings that are duplicated to within a certain tolerance $k$ (usually edit distance or Hamming distance). Moreover, the repeated substring may be subject to other constraints: it may be required to be of at least a certain length, and certain positions in it may be required to be invariant.

Furthermore, efficient algorithms for computing the approximate repetitions are also directly applicable to molecular biology (see [11, 17, 24]) and in particular in DNA sequencing by hybridization ([27]), reconstruction of DNA sequences from known DNA fragments (see [29,30]), in human organ and bone marrow transplantation as well as the determination of evolutionary trees among distinct species ([29]).

Another type of repetition that is used in computer assisted music analysis is that of finding evolutionary chains: given a string $t$ (the "text") and a pattern $p$ ( the "motif"), find whether there exists a sequence $u_{1}=p, u_{2}, \ldots, u_{\ell}$ occurring in the text $t$ such that $u_{i+1}$ occurs to the right of $u_{i}$ in $t$ and $u_{i}$ and $u_{i+1}$ are "similar" for $1 \leq i<\ell$ (i.e. they differ by a certain number of symbols). In [9] and [7] algorithms for overlapping and non-overlapping evolutionary chains were presented and several variants of the problem were studied: computing the longest chain, computing the chain with the least number of errors.

The paper is organised as follows. In the next section we present some basic definitions for strings and background notions for approximate matching. In Section 3 we present an algorithm for $\delta$-approximate (the first notion of approximation) pattern matching. In section 4 we present an algorithm for $\delta, \gamma$-approximate (the second notion of approximation) pattern matching. In section 5 we present algorithms for computing all $\delta$ and $\{\delta, \gamma\}$ - approximate squares in a given text. Finally in Section 6 we present our conclusions and open problems.

## 2 Background and basic string definitions

A string is a sequence of zero or more symbols from an alphabet $\Sigma$; the string with zero symbols is denoted by $\epsilon$. The set of all strings over the alphabet $\Sigma$ is denoted by $\Sigma^{*}$. A string $x$ of length $n$ is represented by $x_{1} \ldots x_{n}$, where $x_{i} \in \Sigma$ for $1 \leq i \leq n$. A string $w$ is a substring of $x$ if $x=u w v$ for $u, v \in \Sigma^{*}$; we equivalently say that the string $w$ occurs at position $|u|+1$ of the string $x$. The position $|u|+1$ is said to be the starting position of $w$ in $x$ and the position $|w|+|u|$ the end position of $u$ in $x$. A string $w$ is a prefix of $x$ if $x=w u$ for $u \in \Sigma^{*}$. Similarly, $w$ is a suffix of $x$ if $x=u w$ for $u \in \Sigma^{*}$.

The string $x y$ is a concatenation of two strings $x$ and $y$. The concatenations of $k$ copies of $x$ is denoted by $x^{k}$. For two strings $x=x_{1} \ldots x_{n}$ and $y=y_{1} \ldots y_{m}$ such that $x_{n-i+1} \ldots x_{n}=y_{1} \ldots y_{i}$ for some $i \geq 1$, the string $x_{1} \ldots x_{n} y_{i+1} \ldots y_{m}$ is a superposition of $x$ and $y$. We say that $x$ and $y$ overlap.

Let $x$ be a string of length $n$. A prefix $x_{1} \ldots x_{p}, 1 \leq p<n$, of $x$ is a period of $x$ if $x_{i}=x_{i+p}$ for all $1 \leq i \leq n-p$. The period of a string $x$ is the shortest period of $x$. A string $y$ is a border of $x$ if $y$ is a prefix and a suffix of $x$.

Let $\Sigma$ be an alphabet of integers and $\delta$ an integer. Two symbols $a, b$ of $\Sigma$ are said to be $\delta$-approximate, denoted $a=\delta b$ if and only if

$$
|a-b| \leq \delta
$$

We say that two strings $x, y$ are $\delta$-approximate, denoted $x \stackrel{\delta}{=} y$ if and only if

$$
\begin{equation*}
|x|=|y|, \text { and } x_{i}={ }_{\delta} y_{i}, \forall i \in\{1 . .|x|\} \tag{2.1}
\end{equation*}
$$

Let $\gamma$ be an integer. Two strings $x, y$ are said to be $\gamma$-approximate, denoted $x \stackrel{\gamma}{=} y$ if and only if

$$
\begin{equation*}
|x|=|y|, \text { and } \sum_{1}^{|x|}\left|x_{i}-y_{i}\right|<\gamma \tag{2.2}
\end{equation*}
$$

Furthermore, we say that two strings $x, y$ are $\{\gamma, \delta\}$-approximate, denoted $x \stackrel{\gamma, \delta}{=} y$, if and only if $x$ and $y$ satisfy conditions (2.1) and (2.2).

## $3 \delta$-Approximate Pattern Matching

The problem of $\delta$-approximate pattern matching is formally defined as follows: given a string $t=t_{1} \ldots t_{n}$ and a pattern $p=p_{1} \ldots p_{m}$ compute all positions $j$ of $t$ such that

$$
p \stackrel{\delta}{=} t[j . . j+m-1]
$$

The algorithm is based on the $O(1)$-time computation of the "Delta states" $D$ State $_{j}, j \in\{1 . . n\}$ by using bit operations under the assumption that $m \leq w$, where $w$ is the number of bits in a machine word. The basic steps of the algorithm are as follows:

1. First we compute the "Delta table" $D T$ : we set $D T(\alpha)=r$, where $\alpha$ denotes a symbol occurring in $t$ and $r=r_{1} \ldots r_{m}$ is a binary word with $r_{i}$ equal to 1 if $\left|\alpha-p_{i}\right| \leq \delta$, otherwise $r_{i}$ is equal to 0 for $i \in\{1 . . m\}$.
2. Let LeftShift be a bit-wise operation that shifts the bits of a binary word by one position to the left. We define

$$
\begin{equation*}
\text { DState }_{j}=\left(\text { LeftShift }^{\left(\text {DState }_{j-1}\right)} \text { OR 1) AND } \quad D T\left[t_{j}\right]\right. \tag{3.1}
\end{equation*}
$$

for $j=1 \ldots n$ and DState $_{0}=0$; hence this procedure is called "Shift-AnD". Once we have computed the $D T$ table, we can use it to compute the $D_{\text {State }}^{j}$ for $j=1 \ldots n$, using the recursive formula (3.1).
3. We say that there is a $\delta$-approximate match (or simply $\delta$-match) at position $j-m+1$ if and only if the $m$-th bit of DState $_{j}$ is 1 or equivalently if and only if $D S t a t e_{j}$, is greater or equal to $2^{m-1}$ when it is viewed as a decimal integer.

Example. For $\Sigma=\{1, \ldots, 9\}$ let us consider $p=3,4,6,2, t=3,4,6,2,8,2,4,5,7,1$ and $\delta=1$. In the preprocessing table, $D T(\alpha)$ denotes the positions where $\left|\alpha-p_{i}\right| \leq$ $\delta$. For example, $D T[3]=1011$ because $\left|3-p_{i}\right| \leq 1$ for $i=1,2,4$.

| $i$ | $p_{i}$ | $D T[1]$ | $D T[2]$ | $D T[3]$ | $D T[4]$ | $D T[5]$ | $D T[6]$ | $D T[7]$ | $D T[8]$ | $D T[9]$ |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 4 | 2 | $\mathbf{1}$ | $\mathbf{1}$ | $\mathbf{1}$ | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 6 | 0 | 0 | 0 | 0 | $\mathbf{1}$ | $\mathbf{1}$ | $\mathbf{1}$ | 0 | 0 |
| 2 | 4 | 0 | 0 | $\mathbf{1}$ | $\mathbf{1}$ | $\mathbf{1}$ | 0 | 0 | 0 | 0 |
| 1 | 3 | 0 | $\mathbf{1}$ | $\mathbf{1}$ | $\mathbf{1}$ | 0 | 0 | 0 | 0 | 0 |

Table 1. The table $D T$ for pattern $p=2,6,4,3$ and alphabet $\Sigma=\{1, \ldots, 9\}$.

The table below evaluates DState $_{j}$ using the relation (3.1). For example,

$$
\begin{aligned}
\text { DState }_{4} & =\left(\text { LeftShift }^{\left(\text {DState }_{3}\right) \text { OR 1) AND } D T\left[t_{4}\right]}\right. \\
& =\left(\text { LeftShift }^{2}(0100)\right. \text { OR 1) AND DT[2] } \\
& =(1000 \text { OR 1) AND 1001 } \\
& =1001 \text { AND 1001 } \\
& =1001
\end{aligned}
$$

which implies that there is a match starting at position 1 of $t$, since the 4 -th bit of $D$ State $_{4}$ is 1 .

| $j$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| $t_{j}$ | 3 | 4 | 6 | 2 | 8 | 2 | 4 | 5 | 7 | 1 |
| LeftShift(DState ${ }_{j-1}$ ) OR 1 | 0001 | 0011 | 0111 | 1001 | 0011 | 0001 | 0011 | 0111 | 1101 | 1001 |
| $D T\left[t_{j}\right]$ | 1011 | 0011 | 0100 | 1001 | 0000 | 1001 | 0011 | 0110 | 0100 | 1000 |
| DState ${ }_{j}$ | 0001 | 0011 | 0100 | 1001 | 0000 | 0001 | 0011 | 0110 | 0100 | 1000 |
| $\left[\text { DState }_{j}\right]_{10}$ | 1 | 3 | 4 | 9 | 0 | 1 | 3 | 6 | 4 | 8 |

Table 2. Computing the Dstates and finding the $\delta$-approximate matches.

A $\delta$-approximate match occurs at position $j-m+1$ of $t$ if $\left[\text { DState }_{j}\right]_{10} \geq 2^{m-1}$, where $\left[\text { DState }_{j}\right]_{10}$ denotes the DState $_{j}$ as a decimal integer. Therefore, there is one match ending at position 4 of $t(\{3,4,6,2\})$ and another one at position 10 of $t(\{4,5,7,1\})$ since $\left\{\right.$ DState $_{4}$, DState $\left._{10}\right\} \geq 2^{3}$.

### 3.1 Pseudo-code

Fig. 1 gives a complete specification of the algorithm. In the line 3 we have the preprocessing phase which compute the $D T$ table. In line 6 we use the recursive formula to compute the DStates. Finally, in line 7 we apply the matching criteria to see whether there is a $\delta$-approximate match or not.

```
procedure \(\operatorname{Shift}-\operatorname{And}(p, t, \delta) \quad\{n=|t|, m=|p|\}\)
begin
    \(D T_{i}[\alpha] \leftarrow\left\{\begin{array}{ll}1 & \text { if }\left|\alpha-p_{i}\right| \leq \delta \\ 0 & \text { otherwise }\end{array} \quad \forall i \in\{1 . . m\}, \forall \alpha \in \Sigma\right.\)
    DState \(_{0} \leftarrow 0\)
    for \(j \leftarrow 1\) to \(n\) do
        DState \(_{j} \leftarrow\left(\right.\) LeftShift \(^{\left.\left(\text {DState }_{j-1}\right) \text { OR } 1\right) \text { AND } D T\left[t_{j}\right]}\)
        if DState \(_{j} \geq 2^{m-1}\) then write \(j-m+1\)
    od
end
```

Fig. 1. The Shift-And Procedure.

### 3.2 Running time

Assuming that the pattern length is no longer than the memory word size of the machine (thus $\mathrm{O}(1)$ size), the time complexity of the preprocessing phase is $O(n)$ (since we need to evaluate DT only for the symbols that occur in $t$ ) and the time complexity of the searching phase in $O(n)$. Figure 2 shows the timing ${ }^{1}$ for different text sizes.


Fig. 2. Timing curves for the Shift-And Procedure.

## $4\{\delta, \gamma\}$-Approximate Pattern Matching

The problem of $\{\delta, \gamma\}$-approximate pattern matching is formally defined as follows: given a string $t=t_{1} \ldots t_{n}$ and a pattern $p=p_{1} \ldots p_{m}$ compute all positions $j$ of $t$ such that

$$
p \stackrel{\delta, \gamma}{=} t[j . . j+m-1]
$$

In order to solve this problem we first make use of the ShifT-AND algorithm to find the $\delta$-approximate matches of the pattern $p$ in $t$. Once we find a $\delta$ approximate match we want to know whether it is also a $\gamma$-approximate match. To do so, we seek to compute successive "Delta States" DState $j_{j}$ and "Gamma States" GStates $j_{j}$ in $O(1)$ time using bit operations under the assumption that $m \leq w$ where $w$ is the number of bits in a machine word. The main steps of the algorithm are as follows:

1. We need to compute the "Delta Table" $D T$ as we did before and the "Gamma Table" $G T$ table; we set $G T(\alpha)=r$, where $\alpha$ denotes a symbol in the alphabet and $r=r_{1} \ldots r_{m}$ is a word with $r_{i}$ equal to $\left|\alpha-p_{i}\right|$ if $\left|\alpha-p_{i}\right| \leq \delta$, otherwise $r_{i}$ is equal to 0 for $i \in\{1 . . m\}$. Each $r_{i}, i \in\{1 . . m\}$ is stored as a binary number of $d$ bits where $d=\lceil\log (\delta \times m)\rceil$.

[^1]2. Let LeftShift be a bit-wise operation that shifts the bits of a binary word one position to the left and RightShift shifts the bits of a binary word $d$ positions to the right. Once we have computed the $D T$ and $G T$ tables, we can use them to compute the DState $_{j}$ and GState $_{j}$ for $j=1 \ldots n$, using the recursive formulas
\[

$$
\begin{gather*}
\text { DState }_{j}=\left(\begin{array}{llll}
\text { LeftShift }^{\left(\text {DState }_{j-1}\right)} \text { OR 1) } & \text { AND } & D T\left[t_{j}\right] \\
\text { GState } \left._{j}={\operatorname{RightShift~}\left(\text { GState }_{j-1}, d\right)+G T} t_{j}\right]
\end{array}, ~\right. \tag{4.1}
\end{gather*}
$$
\]

We also need to define the seeds DState $_{0}=0$ and GState $_{0}=0$. We call this procedure "Shift-Plus" because we use the "shift" and "plus" operators to compute each new state.
3. We say that there is a match ( $\{\delta, \gamma\}$-approximate match) at position $j-m+1$ if and only if the $m$-th bit of DState $_{j}$ is 1 and the $m$-th block of $d$ bits taken as an integer is $\leq \gamma$.

Example. For our example let $\Sigma=\{1, \ldots, 9\}$, the pattern $p=3,4,6,2$, the text $t=3,4,6,2,8,2,4,5,7,1, \delta=1$ and $\gamma=3$. We will use blocks of size 3 $(d=3)$ to store the $\left|\alpha-p_{i}\right|$ values where $\left|\alpha-p_{i}\right| \leq \delta$. For example, $G T[3]=$ 000100000100 because $\left|3-p_{i}\right| \leq 1$ for $\mathrm{i}=1,2,4$ and the differences are $0,1,1$ respectively. (see left hand table of table 3).

| $i$ | $p_{i}$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  |  | 0 | 1 | 0 | 1 | 0 |  | 0 | 0 |  |  |
| 1 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  | 0 |
|  |  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  | 0 |
| 2 |  | 0 | 0 |  | 0 | 1 |  | 0 | 0 |  |  |
|  | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  | 0 |
|  |  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  | 0 |
| 3 |  | 0 | 0 | 0 | 0 | 1 | 0 |  | 0 |  |  |
|  | 6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  | 0 |
|  |  |  |  |  |  | 0 |  |  | 0 |  |  |
| 4 |  | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |  | 0 |
|  | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  | 0 |
|  |  | 0 |  |  |  |  |  |  | 0 |  |  |


| $j$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |
| $t_{j}$ | 3 | 4 | 6 | 2 | 8 | 2 | 4 | 5 | 7 | 1 |
| 3 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
|  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
|  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
|  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 6 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 |
|  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
|  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 1 | 0 | 1 | $\mathbf{0}$ | 1 | 0 | 1 | 0 | 0 | $\mathbf{0}$ |
|  | 0 | 0 | 0 | $\mathbf{0}$ | 0 | 0 | 0 | 0 | 1 | $\mathbf{0}$ |
|  | 0 | 0 | 0 | $\mathbf{0}$ | 0 | 0 | 0 | 0 | 0 | $\mathbf{1}$ |

Table 3. The left hand side table is the "Gamma Table" $G T$ and the right hand side table is the table for finding $\{\gamma, \delta\}$-approximate matches.

The right hand table above shows the computation of the DStates and the GStates using (4.2). For example,

$$
\begin{aligned}
\text { GState }_{9} & =\text { RightShift }(000010010000,3)+000000100000 \\
& =000000010010+000000100000=000000110010
\end{aligned}
$$

We already know that there are two $\delta$-approximate matches ending at positions 4 and 10 of $t$. Now we can use the last three bits of GState ${ }_{4}$ and GState ${ }_{10}$ to find out the values of $\gamma$, which are 0 and 4 respectively (see right hand table of Fig. 3).

### 4.1 Pseudo-code

Fig. 3 below gives a complete description of the algorithm. In the lines 3 and 4 are the preprocessing phase which compute the $D T$ table and $G T$ table respectively. In lines 8 and 9 we compute the next DState and GState respectively. Finally, in line 10 we apply the matching criteria to see whether there is a match or not.

```
procedure \(\operatorname{Shift}-\operatorname{Plus}(p, t, \delta, \gamma) \quad\{n=|t|, m=|p|\}\)
begin
    \(D T_{i}[\alpha] \leftarrow\left\{\begin{array}{ll}1 & \text { if }\left|\alpha-p_{i}\right| \leq \delta \\ 0 & \text { otherwise }\end{array} \forall i \in\{1 . . m\}, \forall \alpha \in \Sigma\right.\)
    \(G T_{d i-d \ldots d i-1}[\alpha] \leftarrow\left\{\begin{array}{l}\left|\alpha-p_{i}\right| \text { if } D T_{i}[\alpha]=1 \\ 0 \text { otherwise }\end{array} \forall i \in\{1 . . m\}, \forall \alpha \in \Sigma\right.\)
    DState \(_{0} \leftarrow 0\)
    GState \(_{0} \leftarrow 0\)
    for \(j \leftarrow 1\) to \(n\) do
        DState \(_{j} \leftarrow\left(\right.\) LeftShift \(^{\left(\text {DState }_{j-1}\right) \text { OR 1) AND }} \operatorname{DT}\left[t_{j}\right]\)
        GState \(_{j} \leftarrow\) RightShift \(^{\left(\text {GState }_{j-1}, d\right)+G T\left[t_{j}\right]}\)
        if DState \(_{j} \geq 2^{m-1}\) AND GState \(_{d m-d \ldots d m-1} \leq \gamma\) then write \(j-m+1\)
    od
end
```

Fig. 3. The Shift-Plus Algorithm.

### 4.2 Running time

Assuming that $\delta \times m \leq 2^{d}-1$ the time complexity of the preprocessing phase is $O(\delta \times m+|\Sigma|)$ and the time complexity of the searching phase in $O(n)$, thus independent from the alphabet size and the pattern length. Figure 4 shows the timing for different text sizes.


Fig. 4. Timing curves for the Shift-Plus Algorithm.

## 5 Computing Approximate Squares

The problem of computing all $\delta$-approximate squares is formally defined as follows: given a string $t=t_{1} \ldots t_{n}$ and an integer $\delta$, compute all positions $j$ of $t$ for which there exists a word $u$ of length $m$ such that

$$
t[j . . j+m] \stackrel{\delta}{=} u \text { and } t[j+m+1 . . j+2 m] \stackrel{\delta}{=} u
$$

where $u$ is said to be the root of the square.
The problem of computing all $\{\delta, \gamma\}$-approximate squares is formally defined as follows: given a string $t=t_{1} \ldots t_{n}$ and two integers $\delta$ and $\gamma$, compute all positions $j$ of $t$ for which there exists a word $u$ of length $m$ such that

$$
t[j . . j+m] \stackrel{\delta, \gamma}{=} u \text { and } t[j+m+1 . . j+2 m] \stackrel{\delta, \gamma}{=} u
$$

where $u$ is said to be the root of the square.
When we look for a square we will run into two possibilities: the root does or does not occur necessarily in the square.

### 5.1 Consider an approximate square such that the root occurs in the square

The diagonal $\operatorname{diag}(i)$ corresponds to the pair of positions $(j, j+i)$ and therefore to the candidates for squares of length $2 i$. There exists an approximate square of length $2 i$ at position $j$ if there exists a run of values not greater than $\delta$ of length at least $i$ on the diagonal $\operatorname{diag}(i)$ starting at position $j$.

For example, consider $\operatorname{diag}(2)$ (see table 4) and $\delta=1$. We are trying to locate runs of length at least 2 containing only values not greater than $\delta=1$. We obtain:

| Position | Square | Roots |
| :--- | :--- | :--- |
| 14 | $(2,3,1,4)$ | $(2,3)$ or $(1,4)$ |

For this example we only have a $\delta$-approximate square starting at position 14 which root can be either $(2,3)$ or $(1,4)$. Note that the roots certainly occur in the square.

### 5.2 Consider an approximate square such that the root does not occur necessarily in the string

We say that there exists an approximate square of length $2 i$ at position $j$ if there exists a run of values not greater than $2 \delta$ of length at least $i$ on the diagonal $\operatorname{diag}(i)$ starting at position $j$. In other words, we are using $2 \delta$ instead of $\delta$.

For example, consider $\operatorname{diag}(2)$ (see table of Fig. 4) and $\delta=1$. We are trying to locate runs of length at least 2 containing only values not greater than $2 \delta=2$. We obtain:


Table 4. Table for computing approximate squares.

| Position | Square | Roots |
| :--- | :--- | :--- |
| 5 | $(-1,-7,1,-5)$ | $(0,-6)$ |
| 9 | $(-5,3,-3,1)$ | $(-4,2)$ |
| 12 | $(1,1,2,3)$ | $(1,2)$ or $(2,2)$ |
| 13 | $(1,2,3,1)$ | $(2,1)$ or $(2,2)$ |
| 14 | $(2,3,1,4)$ | $(1,3),(1,4),(2,3)$ or $(2,4)$ |

Furthermore, consider $\operatorname{diag}(3)$ and $\delta=1$. We are trying to locate runs of length at least 3 containing only values not greater than $2 \delta=2$. We obtain:

| Position | Square | Roots |
| :--- | :--- | :--- |
| 1 | $(2,-3,-5,4,-1,-7)$ | $(3,-2,-6)$ |
| 6 | $(-7,1,-5,-5,3,-3)$ | $(-6,2,-4)$ |
| 12 | $(1,1,2,3,1,4)$ | $(2,0,3),(2,1,3)$ or $(2,2,3)$ |
| 13 | $(1,2,3,1,4,5)$ | $(0,3,4),(1,3,4)$ or $(2,3,4)$ |

In those cases where we want to consider a $\{\delta, \gamma\}$-approximate square we just check each $\delta$-approximate match to see if it is also a $\{\delta, \gamma\}$-approximate square.

In the last example we will like to consider $\delta=1$ and $\gamma=4$. This means that we are trying to locate runs of length at least 2 containing only values not greater than $2 \delta=2$ but with $\gamma \leq 4$. We obtain:

| Position | Square | Roots | $\gamma$ |
| :--- | :--- | :--- | :--- |
| 12 | $(1,1,2,3,1,4)$ | $(2,0,3),(2,1,3)$ or $(2,2,3)$ | 4 |
| 13 | $(1,2,3,1,4,5)$ | $(0,3,4),(1,3,4)$ or $(2,3,4)$ | 4 |

### 5.3 Pseudo-code

Fig. 5 gives the algorithm that solves the $\delta$-approximation square problem. Fig. 6 below gives the algorithm that solves the $\{\delta, \gamma\}$-approximation square problem.

```
procedure DeltaSquares \((t, \delta) \quad\{n=|t|\}\)
begin
    for \(\operatorname{diag} \leftarrow 2\) to \(n / 2\) do
            \(\mathrm{i} \leftarrow 0 ;\) dsum \(\leftarrow 0\)
            for \(j \leftarrow \operatorname{diag}\) to \(n\) do
                \(\operatorname{diff} \leftarrow|t[i]-t[j]|\)
                if \(\operatorname{diff} \leq \delta\) then \(d\) sum \(\leftarrow d\) sum +1
                else dsum \(\leftarrow 0\)
                if dsum \(\geq \operatorname{diag}\) then write \(j-2 * \operatorname{diag}+2\)
                \(i \leftarrow i+1\)
            od
        od
end
```

Fig. 5. The DeltaSquares Algorithm.

```
procedure DeltaGammaSquares \((t, \delta, \gamma) \quad\{n=|t|\}\)
begin
    for \(\operatorname{diag} \leftarrow 2\) to \(n / 2\) do
        \(\mathrm{i} \leftarrow 0 ;\) dsum \(\leftarrow 0 ;\) gsum \(\leftarrow 0\)
        for \(j \leftarrow \operatorname{diag}\) to \(n\) do
                \(\operatorname{diff} \leftarrow|t[i]-t[j]|\)
                if \(\operatorname{diff} \leq \delta\) then
                begin
                    dsum \(\leftarrow\) dsum +1
                    gsum \(\leftarrow\) gsum + diff
                    if dsum \(>\) diag then \(g s u m \leftarrow g s u m-|t[i-\operatorname{diag}]-t[j-\operatorname{diag}]|\)
                end
                else
                begin
                    dsum \(\leftarrow 0 ;\) gsum \(\leftarrow 0\)
                end
                if \(d s u m \geq \operatorname{diag}\) AND \(g s u m \leq g\) then write \(j-2 * \operatorname{diag}+2\)
                \(i \leftarrow i+1\)
            od
    od
end
```

Fig. 6. The DeltaGammaSquares Algorithm.

### 5.4 Running time

The complexity of these algorithms is easily seen to be $O\left(n^{2}\right)$. Figure 7 shows the timing for different text sizes.


Fig. 7. Timing curves for $\{\delta, \gamma\}$-approximate squares.

## 6 Conclusion and Open problems

The running time of the computation of $\delta$-approximate squares can be reduced to $O(n \log n)$; A theoretical algorithm is presented in [16] that shadows the Main and Lorentz algorithm ([21]).

The following two problems are still open:
Problem 1. Given a string $t=t_{1} \ldots t_{n}$ and two integers $m$ and $\delta$, compute all positions $j$ of $t$, that there exists a string $\hat{t}$ such that

$$
\begin{gathered}
t[j . . j+m] \stackrel{\delta}{=} \hat{t} \\
t[j+m+1 . . j+2 m] \stackrel{\delta}{=} \hat{t} \\
\ldots \\
t[j+\ell m+1 . . j+(\ell+1) m] \stackrel{\delta}{=} \hat{t}
\end{gathered}
$$

Problem 2 Given a string $t=t_{1} \ldots t_{n}$ and three integers $m, \delta$ and $\gamma$, compute all positions $j$ of $t$, that there exists a string $\hat{t}$ such that

$$
\begin{gathered}
t[j . . j+m] \stackrel{\delta, \gamma}{=} \hat{t} \\
t[j+m+1 . . j+2 m] \stackrel{\delta, \gamma}{=} \hat{t} \\
\ldots \\
t[j+\ell m+1 . . j+(\ell+1) m] \stackrel{\delta, \gamma}{=} \hat{t}
\end{gathered}
$$

## References

1. A. Apostolico and F. P. Preparata, Optimal Off-line Detection of Repetitions in a String, Theoretical Computer Science, 22 (3) 297-315 (1983).
2. R. A. Baeza-Yates and G. H. Gonnet, A new approach to text searching, CACM, Vol 35, (1992), pp. 74-82.
3. O. Berkman, C. Iliopoulos and K. Park, String covering, Information and Computation 123 (1996), pp. 127-137.
4. E. Cambouropoulos, T. Crawford and C.S. Iliopoulos, (1999) Pattern Processing in Melodic Sequences: Challenges, Caveats and Prospects. In Proceedings of the AISB'99 Convention (Artificial Intelligence and Simulation of Behaviour), Edinburgh, U.K., pp. 42-47 (1999).
5. D. Cope, Pattern-Matching as an Engine for the Computer Simulation of Musical Style, In Proceedings of the International Computer Music Conference, Glasgow, pp.288-291 (1990).
6. T. Crawford, C. S. Iliopoulos, R. Raman, String Matching Techniques for Musical Similarity and Melodic Recognition, Computing in Musicology, Vol 11 (1998) 73100.
7. T. Crawford, C.S. Iliopoulos, R. Winder, H. Yu, Approximate musical evolution, in the Proceedings of the 1999 Artificial Intelligence and Simulation of Behaviour Symposium (AISB'99), G. Wiggins (ed), The Society for the Study of Artificial Intelligence and Simulation of Behaviour, Edinburgh, 76-81,(1999).
8. M. Crochemore, An optimal algorithm for computing the repetitions in a word, Information Processing Letters 12 (1981), pp. 244-250.
9. M. Crochemore, C.S. Iliopoulos and H. Yu, Algorithms for computing evolutionary chains in molecular and musical sequences, Proceedings of the 9-th Australasian Workshop on Combinatorial Algorithms Vol 6 (1998) 172-185.
10. A. Czumaj, P. Ferragina, L. Gasieniec, S. Muthukrishnan and J. Traeff, The architecture of a software library for string processing, to be presented at Workshop on Algorithm Engineering, Venice, September 1997.
11. V. Fischetti, G. Landau, J.Schmidt and P. Sellers, Identifying periodic occurences of a template with applications to protein structure, Proc. 3rd Combinatorial Pattern Matching, Lecture Notes in Computer Science, vol. 644, 1992, pp. 111120.
12. Z. Galil and K. Park, An improved algorithm for approximate string matching, SIAM Journal on Computing, 19 (1990), pp. 989-999.
13. C. S. Iliopoulos and L. Mouchard, An $O(n \log n)$ algorithm for computing all maximal quasiperiodicities in strings, to appear in the Proceedings of CATS'99: "Computing: Australasian Theory Symposium", Auckland, New Zealand, Lecture Notes in Computer Science, Springer Verlag, Vol 213 (1999) 262-272.
14. C. S. Iliopoulos, D. W. G. Moore and K. Park, Covering a string, Algorithmica 16 (1996), pp. 288-297.
15. C. S. Iliopoulos, D. W. G. Moore and W. F. Smyth, A linear algorithm for computing the squares of a Fibonacci string, in P. Eades and M. Moule, eds. Proceedings CATS'96, "Computing: Australasian Theory Symposium," University of Melbourne, pp. 55-63, 1996.
16. C.S. Iliopoulos and Y.J. Pinzon, An $O(n \log n)$ algorithm for computing squares in musical sequences, in preparation.
17. S. Karlin, M. Morris, G. Ghandour, and M.-Y. Leung, Efficients algorithms for molecular sequences analysis, Proc. Natl. Acad. Sci., USA (1988) 85:841-845
18. G. M. Landau and J. P. Schmidt, An algorithm for approximate tandem repeats, in Proc. Fourth Symposium on Combinatorial Pattern Matching, Springer-Verlag Lecture Notes in Computer Science 648, pp. 120-133, 1993.
19. G.M. Landau and U. Vishkin, Introducing efficient parallelism into approximate string matching and a new serial algorithm, in Proc. Annual ACM Symposium on Theory of Computing, ACM Press, pp. 220-230, 1986.
20. G.M. Landau and U. Vishkin, Fast string matching with $k$ differences, Journal of Computer and Systems Sciences, 37 (1988), pp. 63-78.
21. G. Main and R. Lorentz, An $O(n \log n)$ algorithm for finding all repetitions in a string, Journal of Algorithms 5 (1984), pp. 422-432.
22. M. Mongeau and D. Sankoff, Comparison of Musical Sequences, Computers and the Humanities 24 (1990), pp. 161-175.
23. P. McGettrick, MIDIMatch: Musical Pattern Matching in Real Time. MSc Dissertation, York University, U.K. (1997).
24. A. Milosavljevic and J. Jurka, Discovering simple DNA sequences by the algorithmic significance method, Comput. Appl. Biosci. (1993) 9:407-411
25. D. W. G. Moore and W. F. Smyth, Computing the covers of a string in linear time, in Proc. 5th ACM-SIAM Symposium on Discrete Algorithms, pp. 511-515, 1994.
26. E. W. Myers and S. K. Kannan, An algorithm for locating non-overlapping regions of maximum alignment score, in Proc. Fourth Symposium on Combinatorial Pattern Matching, Springer-Verlag Lecture Notes in Computer Science 648, pp. 74-86, 1993.
27. P. A. Pevzner \& W. Feldman, Gray Code Masks for DNA Sequencing by Hybridization, Genomics, 23, 233-235 (1993).
28. P.Y. Rolland, J.G. Ganascia, Musical Pattern Extraction and Similarity Assessment. In Readings in Music and Artificial Intelligence. E. Miranda. (ed.). Harwood Academic Publishers (forthcoming) (1999).
29. J. P. Schmidt, All shortest paths in weighted grid graphs and its application to finding all approximate repeats in strings, in Proc. of the Fifth Symposium on Combinatorial Pattern Matching CPM'94, Lecture Notes in Computer Science (1994).
30. S. S. Skiena \& G. Sundaram, Reconstructing strings from substrings, J. Computational Biol. 2 (1995) 333-353.
31. S. Wu and U. Manber, Fast text searching allowing errors, CACM, Vol 35, (1992), pp. 83-91.


Pitch Interval Pattern: $\{1,-1,3,0\}$
Pitch Interval String:
$\{1,-1,3,0,-5,2,-2,3,0,-5,0,2,0,2,3,-2,-1,-2,2,1,-1,3,0,-5,2,-2,3,0,-5,2,2,1,-1,-2,-2,7,2,-2,2,0,3,-1,-2,0$, $-2,0,0,-3,-4,7,-2,-3,5,-3,-4,4,-2,2,1,-1,3,0,-5,2,-2,3,0,-5,2,2,1,-1,-2,0,2,0,1,2,2,2,1,-12,4,-2,10,-9\}$

## APPENDIX I

Melody from Mozart's Sonata in A major
This melody may be represented as a string of pitch intervals (in number of semitones). If exact matching is employed, three identical instances of the search pattern are found (patterns $a, c$ and $f$ ); the other 4 instances are not matched. If $\delta$-approximate matching is employed for $\delta=1$, then all seven instances depicted above are found.


Pitch Interval Pattern: $\{5,-1,1,4,3,5,0\}$
Pitch Interval String:
$\{5,-1,1,4,3,5,0,-1,-2,-2,5,-10,2,1,4,-9,2,2,3,-5,-7,5,-1,1,4,3,9,0,-2,-2,-1,1,4,-7,3,-1,-1,-1,2,-4,-12$,
$5,-1,1,4,3,3,0,-1,-2,-2,4,-7,2,1,-1,-2,-5,3,5,-1,1,4,3,5,0,-1,-2,-2,4,-7,2,1,-1,-2,-5,-2,-7$,
$5,-1,1,4,3,5,0,-1,-2,-2,5,-10,2,1,4,-9,2,2,3,-5,-7,5,-1,1,4,3,9,0,-2,-2,-3,-2,5,-10,2,1,4,-7,2,1,4,-12,2,1\}$

## APPENDIX II <br> Melody from Schumann's Träumerei

This melody may be represented as a string of pitch intervals (in number of semitones). If exact matching is employed only 3 identical instances of the given pattern are found (patterns $a, d$ and $e$ ); the other 3 instances are not matched. If $\delta$-approximate matching is employed for $\delta=2$, then 4 instances are found (patterns $a, c, d$ and $e$ ); for $\delta=4$ all 6 instances depicted above are discovered ( $\gamma$ approximate matching may be additionally applied to restrict $\delta$-approximate matching especially for larger $\delta$ values and for larger melodic corpuses.


[^0]:    * Supported by the START programme Y99-INF, Austrian Federal Ministry of Science and Transport
    ** Partially supported by the C.N.R.S. Program "Génomes"
    *** Partially supported by the EPSRC grant GR/J 17844.
    $\dagger$ Partially supported by the C.N.R.S. Program "Génomes"

[^1]:    ${ }^{1}$ Using a SUN Ultra Enterprise 300 MHz running Solaris Unix.

