Approximate String Matching in Musical Sequences
Maxime Crochemore, Costas S. Iliopoulos, Thierry Lecroq, Yoan J. Pinzon

To cite this version:

HAL Id: hal-00619981
https://hal-upec-upem.archives-ouvertes.fr/hal-00619981
Submitted on 19 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.
Approximate String Matching in Musical Sequences

Maxime Crochemore\textsuperscript{1}, Costas S. Iliopoulos\textsuperscript{2}\textsuperscript{†},
Thierry Lecroq\textsuperscript{3} and Y. J. Pinzon\textsuperscript{2}\textsuperscript{‡}

\textsuperscript{1} Institut Gaspard-Monge, Université de Marne-la-Vallée, France.
mac@univ-mlv.fr,

\textsuperscript{2} Dept. of Computer Science, King’s College London, London WC2R 2LS, England, and School of Computing, Curtin University of Technology, GPO Box 1987 U, WA.

Australia

\{csi,pinzon\}@dcs.kcl.ac.uk,

\textsuperscript{3} LIFAR - ABISS, Université de Rouen, 76821 Mont Saint Aignan Cedex, France.
lecroq@dir.univ-rouen.fr

Abstract. Here we consider computational problems on \(\delta\)-approximate and
\((\delta, \gamma)\)-approximate string matching. These are two new notions of approximate
matching that arise naturally in applications of computer assisted music analysis.
We present fast, efficient and practical algorithms for these two notions of
approximate string matching.

Key words: String algorithms, approximate string matching, dynamic program-
ing, 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 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.

The approximate repetition problem has been extensively studied over the last few
years. Efficient algorithms for computing the approximate repetitions are directly appli-
cable to molecular biology (see [7, 9, 12]) and in particular in DNA sequencing by

\textsuperscript{*}This work was partially supported by a NATO grant PST.CLG.977017.

\textsuperscript{†}Partially supported by a Marie Curie fellowship, Wellcome and Royal Society grants.

\textsuperscript{‡}Partially supported by an ORS studentship and EPSRC Project GR/L92150.
hybridization ([13]), reconstruction of DNA sequences from known DNA fragments (see [15, 16]), in human organ and bone marrow transplantation as well as the determination of evolutionary trees among distinct species ([15]).

The approximate matching problem has been used for a variety of musical applications (see overviews in McGettrick [11]; Crawford et al [6]; Rolland et al [14]; Cambourpoulos et al [3]). It is known that exact matching cannot be used to find occurrences of a particular melody. Approximate matching should be used in order to allow the presence of errors. The number of errors allowed will be referred to as $\delta$. This paper focuses in one special type of approximation that arise especially in musical information retrieval, i.e. $\delta$-approximation. 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).

In [4], a number of efficient algorithms for $\delta$-approximate matching were presented (i.e. the Shift-AND algorithm and Shift-PLUS algorithm). The Shift-AND algorithm is based on the $O(1)$-time computation of different states for each symbol in the text. Hence the overall complexity is $O(n)$. These algorithms use the bitwise technique. It is possible to adapt fast and practical exact pattern matching algorithms to these kind of approximations. In this paper we will present the adaptations of the Tuned-Boyer-Moore [8], the Skip-Search algorithm [5] and the Maximal-Shift algorithm [17] and present some experiments to assert that these adaptations are faster than the algorithms using the bitwise technique.

The paper is organised as follows. In the next section we present some basic definitions for strings and background notions for approximate matching. In Sections 3-5 we present the adaptation of Tuned-Boyer-Moore, Skip-Search and Maximal-Shift algorithms to speed-up $\delta$-approximate pattern matching algorithms and in section 6 to speed-up ($\delta, \gamma$)-approximate pattern matching algorithms. In section 7 we present the experimental results of these algorithms. Finally in Section 8 we present our conclusions.

## 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 $x = uvw$ 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 \( w \) in \( x \). A string \( w \) is a prefix of \( x \) if \( x = wu \) for \( u \in \Sigma^* \). Similarly, \( w \) is a suffix of \( x \) if \( x = uw \) for \( u \in \Sigma^* \).

The string \( xy \) is a concatenation of two strings \( x \) and \( y \). The concatenation 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 \( n \). The integer \( p \) is said to be 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 smallest 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 \( a \equiv \delta b \) if and only if

\[
|a - b| \leq \delta
\]

We say that two strings \( x, y \) are \( \delta \)-approximate, denoted \( x \equiv \delta y \) if and only if

\[
|x| = |y|, \text{ and } x_i \equiv \delta y_i, \forall i \in \{1, \ldots, |x|\}
\]  

(2.1)

For a given integer \( \gamma \) we say that two strings \( x, y \) are \( \gamma \)-approximate, denoted \( x \equiv \gamma y \) if and only if

\[
|x| = |y|, \text{ and } \sum_{i=1}^{|x|} |x_i - y_i| \leq \gamma
\]  

(2.2)

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

### 3 \( \delta \)-Tuned-Boyer-Moore 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

\[
p \equiv \delta t[j..j + m - 1]
\]

A naive solution of this problem is to build an Aho-Corasick automaton (see [1]) of all strings that are \( \delta \)-approximate to \( p \) and then use the automaton to process \( t \). The time required to build the automaton is \( O(|\Sigma|^\delta) \), thus this method is of no practical use as e.g. we can have \( |\Sigma| \approx 180 \) and \( |\delta| \approx 10 \). In [4] an efficient algorithm was presented based on the \( O(1) \)-time computation of the “delta states” by using bit operations under the assumption that \( m \leq w \), where \( w \) is the number of bits in a machine word.

Here we present an adaptation of the Tuned-Boyer-Moore for exact pattern matching algorithm to \( \delta \)-approximate pattern matching. The exact pattern matching problem consists in finding one or more (generally all) exact occurrences of a pattern \( p \) of length \( m \) in a text \( t \) of length \( n \). Basically a pattern matching algorithm uses a window which size is equal to the length of the pattern. It first aligns the left ends
of the window and the text. Then it checks if the pattern occurs in the window and
shifts the window to the right. It repeats the same procedure again until the right
end of the window goes beyond the right end of the text.

The Tuned-Boyer-Moore algorithm is a very fast practical variant of the fa-
mous Boyer-Moore algorithm [2]. It only uses the occurrence shift function to
perform the shifts. The occurrence shift function is defined for each symbol \( a \) in the
alphabet \( \Sigma \) as follows:

\[
\text{shift}[a] = \min\{\{m - i \mid p_i = a\} \cup \{m\}\}
\]

The Tuned-Boyer-Moore algorithm gains its efficiency by unrolling three
shifts in a very fast skip loop to locate the occurrences of the rightmost symbol
of the pattern in the text. Once an occurrence of \( p_m \) is found, it checks naively if the
whole pattern occurs in the text. Then the shift consists in aligning the rightmost
symbol of the window with the rightmost reoccurrence of \( p_m \) in \( p_1 \ldots p_{m-1} \), if any.
The length \( s \) of this shift is defined as follows:

\[
s = \min\{\{m - i \mid p_i = p_m \text{ and } i > 0\} \cup \{m\}\}
\]

To do \( \delta \)-approximate pattern matching, the shift function can be defined to be for
each symbol \( a \) in the alphabet \( \Sigma \) the distance from the right end of the pattern of the
closest symbol \( p_i \) such that \( p_i \delta \equiv a \):

\[
\text{shift}[a] = \min\{\{m - i \mid p_i \equiv a\} \cup \{m\}\}
\]

Then the length of the shift \( s \) becomes:

\[
s = \min\{\{m - i \mid p_i \equiv p_m \text{ and } i > 0\} \cup \{m\}\}
\]

The pseudo-code for \( \delta \)-Tuned-Boyer-Moore algorithm can be found in Fig-
ure 1.

4 \( \delta \)-Skip-Search Approximate Pattern Matching

In the Skip-Search algorithm, for each symbol of the alphabet, a bucket collects
all of that symbol’s positions in \( p \). When a symbol occurs \( k \) times in the pattern,
there are \( k \) corresponding positions in the symbol’s bucket. When the word is much
shorter than the alphabet, many buckets are empty. The buckets are stored in a table
\( z \) defined as follows:

\[
z[a] = \{i \mid p_i = a\}
\]

The main loop of the search phase consists of examining every \( m \)th text symbol,
\( t_j \) (so there will be \( n/m \) main iterations). Then for \( t_j \), it uses each position in the
bucket \( z[t_j] \) to obtain a possible starting point of \( p \) in \( t \) and checks if the pattern
occurs at that position.

To do \( \delta \)-approximate pattern matching, the buckets can be computed as follows:

\[
z[a] = \{i \mid p_i \equiv a\}
\]

Figure 2 shows the pseudo-code for \( \delta \)-Skip-Search algorithm.
\begin{figure}[h]
\begin{center}
\begin{minipage}{.8\textwidth}
\begin{algorithmic}[1]
\STATE $\delta$-TUNED-BOYER-MOORE($p, m, t, n, \delta$)
\STATE $\triangleright$ Preprocessing
\FOR{all $a \in \Sigma$
\STATE $\text{shift}[a] \leftarrow \min \{ \{ m - i \mid p_i \trianglelefteq a \} \cup \{ m \} \}$
\STATE $s \leftarrow \min \{ \{ m - i \mid p_i \trianglerighteq p_m \} \cup \{ m \} \}$
\STATE $t_n \ldots t_{n+m-1} \leftarrow (p_m)^m$
\STATE $\triangleright$ Searching
\STATE $j \leftarrow m$
\WHILE{$j \leq n$
\STATE $k \leftarrow \text{shift}[t_j]$
\WHILE{$k \neq 0$
\STATE $j \leftarrow j + k$
\STATE $k \leftarrow \text{shift}[t_j]$
\STATE $j \leftarrow j + k$
\STATE $k \leftarrow \text{shift}[t_j]$
\STATE $j \leftarrow j + k$
\STATE $k \leftarrow \text{shift}[t_j]$
\IF{$p_1 \ldots p_{m-1} \trianglelefteq t_{j-m+1} \ldots t_{j-1}$ and $j \leq n$
\STATE $\text{REPORT}(j - m + 1)$
\ENDIF
\STATE $j \leftarrow j + s$
\ENDWHILE
\ENDWHILE
\end{algorithmic}
\end{minipage}
\end{center}
\end{figure}

Figure 1: Adaptation of the TUNED-BOYER-MOORE exact pattern matching algorithm to do $\delta$-approximate pattern matching.

5 \textbf{\textit{$\delta$-Maximal-Shift Approximate Pattern Matching}}

Sunday [17] designed an exact string matching algorithm where the pattern positions are scanned from the one which will lead to a larger shift to the one which will lead to a shorter shift, in case of a mismatch. Doing so one may hope to maximize the lengths of the shifts and thus to minimize the overall number of comparisons.

Formally we define a permutation

$$\sigma : \{1, 2, \ldots, m, m+1\} \rightarrow \{1, 2, \ldots, m, m+1\}$$

and a function $\text{shift}$ such that

$$\text{shift}[\sigma(i)] \trianglelefteq \text{shift}[\sigma(i+1)]$$

for $1 \leq i < m$ and

$$\text{shift}[\sigma(i)] = \min \{ \ell \mid \text{for } 1 \leq j < i, p_{\sigma(j) - \ell} = p_{\sigma(j)} \text{ and } p_{\sigma(i) - \ell} \neq p_{\sigma(i)} \}$$

for $1 \leq i \leq m$ and $\sigma(m+1) = m+1$. Furthermore $\text{shift}[m+1]$ is set with the value of the period of the pattern $p$.

We also define a function $bc$ for each symbol of the alphabet:
\[ \delta - \text{SKIP-SEARCH}(p, m, t, n, \delta) \]

1. **Preprocessing**
2. **for** all \( a \in \Sigma \)
3. \( \text{do } z[a] \leftarrow \{ i \mid p_i \hat{=} a \} \)
4. **Searching**
5. \( j \leftarrow m \)
6. **while** \( j \leq n \)
7. **do** for all \( i \in z[t_j] \)
8. **do** if \( p_{\hat{=} j} \ldots t_{j + m - 1} \)
9. **then** \( \text{REPORT}(j - i) \)
10. \( j \leftarrow j + m \)

Figure 2: Adaptation of the \text{SKIP-SEARCH} exact pattern matching algorithm to do \( \delta \)-approximate pattern matching.

\[
bc[a] = \begin{cases} 
\min\{ j \mid 0 \leq j < m \text{ and } p_{m-j} = a \} , & \text{if } a \text{ occurs in } p \\
 m , & \text{otherwise} 
\end{cases}
\]

for \( a \in \Sigma \).

Then, when the pattern is aligned with the \( t[j..j + m - 1] \) the comparisons are performed in the following order \( \sigma(1), \sigma(2), \ldots, \sigma(m) \) until the whole pattern is scanned or a mismatch is found. If a mismatch is found when comparing \( p[\sigma(i)] \) then a shift of length \( \max\{\text{shift}[\sigma(i)], bc[t[j + m + 1]]\} \) is performed. Otherwise an occurrence of the pattern is found and the length of the shift is equal to the maximum value between the period of the pattern and \( bc[t[j + m + 1]] \). Then the comparisons resume with \( p_{\sigma(i)} \) without keeping any memory of the comparisons previously done.

To perform \( \delta \)-approximate string matching the two functions can be redefined as follows:

\[
\text{shift}[\sigma(i)] = \min\{ \ell \mid 1 \leq j < i, p_{\sigma(j) - \ell} = 2\delta p_{\sigma(j)} \text{ and } p_{\sigma(j) - \ell} \neq \delta p_{\sigma(i)} \}\]

for \( 1 \leq i \leq m \) and

\[
\text{shift}[m + 1] = \min\{ \ell \mid p[i] = 2\delta p[i + \ell] \text{ for } 1 \leq i \leq m - \ell \}
\]

and

\[
bc[a] = \begin{cases} 
\min\{ j \mid 0 \leq j < m \text{ and } p_{m-j} = \delta a \} , & \text{if such a } j \text{ exists} \\
 m , & \text{otherwise} 
\end{cases}
\]

for \( a \in \Sigma \).

The preprocessing phase can be done in \( O(m^2) \). Figure 3 gives the pseudo-code of the searching phase.
\[ \delta\text{-MAXIMAL-SHIFT}(p, m, t, n, \delta) \]

1. \textbf{Searching}
2. \( j \leftarrow 0 \)
3. while \( j \leq n - m \)
4. \quad do \( i \leftarrow 1 \)
5. \quad \quad while \( i \leq m \) and \( p[\sigma(i)] = t[j + \sigma(i)] \)
6. \quad \quad do \( i \leftarrow i + 1 \)
7. \quad \quad if \( i > m \)
8. \quad \quad \textbf{REPORT}(j)
9. \quad \quad \textbf{MAXIMAL-SHIFT}\]

Figure 3: Adaptation of the \textbf{MAXIMAL-SHIFT} exact pattern matching algorithm to do \( \delta \)-approximate pattern matching.

### 6 \((\delta, \gamma)\)-Approximate String Matching Algorithms

The problem of \((\delta, \gamma)\)-\textit{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

\[ p_{\delta \gamma} = t[j..j + m - 1] \]

In [4] this problem was solved by making use of the \textbf{SHIFT-AND} algorithm to find the \( \delta \)-approximate matches of the pattern \( p \) in \( t \). Once a \( \delta \)-approximate match was found, it was then tested to check whether it is also a \( \gamma \)-approximate match. This was done by computing successive “delta states” and “gamma states” 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.

In order to adapt the \( \delta \)-\textbf{TUNED-BOYER-MOORE}, \( \delta \)-\textbf{SKIP-SEARCH} and \( \delta \)-\textbf{MAXIMAL-SHIFT} algorithms to the case of \((\delta, \gamma)\)-approximation, it just suffices to adapt the naive check of the pattern. The resulting algorithms are named \( (\delta, \gamma)\)-\textbf{TUNED-BOYER-MOORE} algorithm, \( (\delta, \gamma)\)-\textbf{SKIP-SEARCH} algorithm and \( (\delta, \gamma)\)-\textbf{MAXIMAL-SHIFT} algorithm.

### 7 Experimental results

We implemented in C, in a homogeneous way, the following algorithms: \textbf{SHIFT-AND}, \( \delta \)-\textbf{TUNED-BOYER-MOORE}, \( \delta \)-\textbf{SKIP-SEARCH}, \( \delta \)-\textbf{MAXIMAL-SHIFT}, \textbf{SHIFT-PLUS}, \( (\delta, \gamma)\)-\textbf{TUNED-BOYER-MOORE}, \( (\delta, \gamma)\)-\textbf{SKIP-SEARCH} and \( (\delta, \gamma)\)-\textbf{MAXIMAL-SHIFT}.

We randomly built a text of 500k symbols on an alphabet of size \( |\Sigma| = 70 \). We then searched for each values 100 patterns and took the average running time. Times are measured in hundredth of seconds and include both preprocessing and searching times.

The results for \( \delta \)-approximation are shown in tables 1 to 5. For the values used in these experiments, the \( \delta \)-\textbf{TUNED-BOYER-MOORE} algorithm is always faster than the \( \delta \)-\textbf{SKIP-SEARCH} algorithm which is itself always faster than the \textbf{SHIFT-AND} algorithm.
<table>
<thead>
<tr>
<th>$m$</th>
<th>Shift-And</th>
<th>$\delta$-Tuned-Boyer-Moore</th>
<th>$\delta$-Skip-Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>32.98</td>
<td>10.78</td>
<td>18.61</td>
</tr>
<tr>
<td>9</td>
<td>32.90</td>
<td>10.55</td>
<td>18.11</td>
</tr>
<tr>
<td>10</td>
<td>32.93</td>
<td>10.10</td>
<td>17.65</td>
</tr>
<tr>
<td>20</td>
<td>32.86</td>
<td>9.32</td>
<td>15.81</td>
</tr>
</tbody>
</table>

Table 1: Running times for $\delta$-approximation with $\delta = 5$.

<table>
<thead>
<tr>
<th>$m$</th>
<th>Shift-And</th>
<th>$\delta$-Tuned-Boyer-Moore</th>
<th>$\delta$-Skip-Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>33.07</td>
<td>13.40</td>
<td>21.66</td>
</tr>
<tr>
<td>9</td>
<td>32.90</td>
<td>13.00</td>
<td>20.94</td>
</tr>
<tr>
<td>10</td>
<td>32.93</td>
<td>12.64</td>
<td>20.49</td>
</tr>
<tr>
<td>20</td>
<td>32.92</td>
<td>11.97</td>
<td>18.81</td>
</tr>
</tbody>
</table>

Table 2: Running times for $\delta$-approximation with $\delta = 6$.

<table>
<thead>
<tr>
<th>$m$</th>
<th>Shift-And</th>
<th>$\delta$-Tuned-Boyer-Moore</th>
<th>$\delta$-Skip-Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>33.65</td>
<td>16.65</td>
<td>24.99</td>
</tr>
<tr>
<td>9</td>
<td>33.14</td>
<td>16.05</td>
<td>24.06</td>
</tr>
<tr>
<td>10</td>
<td>33.05</td>
<td>15.71</td>
<td>23.62</td>
</tr>
<tr>
<td>20</td>
<td>32.93</td>
<td>14.82</td>
<td>21.42</td>
</tr>
</tbody>
</table>

Table 3: Running times for $\delta$-approximation with $\delta = 7$.

<table>
<thead>
<tr>
<th>$m$</th>
<th>Shift-And</th>
<th>$\delta$-Tuned-Boyer-Moore</th>
<th>$\delta$-Skip-Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>34.72</td>
<td>21.18</td>
<td>29.15</td>
</tr>
<tr>
<td>9</td>
<td>33.41</td>
<td>20.03</td>
<td>27.64</td>
</tr>
<tr>
<td>10</td>
<td>33.07</td>
<td>19.12</td>
<td>26.85</td>
</tr>
<tr>
<td>20</td>
<td>32.81</td>
<td>18.20</td>
<td>24.41</td>
</tr>
</tbody>
</table>

Table 4: Running times for $\delta$-approximation with $\delta = 8$.

The results for $(\delta, \gamma)$-approximation are shown in tables 6 to 10. For the values used in these experiments, the $(\delta, \gamma)$-Tuned-Boyer-Moore algorithm is always faster than the $(\delta, \gamma)$-Skip-Search algorithm which is itself always faster than the Shift-Plus algorithm.

Experiments conducted only on $\gamma$-approximation show that an adaptation to this case of the Skip-Search algorithm is faster than an adaptation of the Tuned-Boyer-Moore algorithm.
<table>
<thead>
<tr>
<th>$m$</th>
<th>Shift-And</th>
<th>$\delta$-Tuned-Boyer-Moore</th>
<th>$\delta$-Skip-Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>36.46</td>
<td>26.82</td>
<td>34.64</td>
</tr>
<tr>
<td>9</td>
<td>34.46</td>
<td>24.36</td>
<td>31.46</td>
</tr>
<tr>
<td>10</td>
<td>33.41</td>
<td>23.61</td>
<td>30.55</td>
</tr>
<tr>
<td>20</td>
<td>33.00</td>
<td>22.32</td>
<td>27.54</td>
</tr>
</tbody>
</table>

Table 5: Running times for $\delta$-approximation with $\delta = 9$.

<table>
<thead>
<tr>
<th>$m$</th>
<th>Shift-Plus</th>
<th>($\delta, \gamma$)-Tuned-Boyer-Moore</th>
<th>($\delta, \gamma$)-Skip-Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>50.73</td>
<td>23.33</td>
<td>31.93</td>
</tr>
<tr>
<td>9</td>
<td>50.32</td>
<td>27.78</td>
<td>35.52</td>
</tr>
<tr>
<td>10</td>
<td>51.79</td>
<td>33.76</td>
<td>39.45</td>
</tr>
<tr>
<td>20</td>
<td>50.26</td>
<td>32.46</td>
<td>36.91</td>
</tr>
</tbody>
</table>

Table 6: Running times for ($\delta, \gamma$)-approximation with $\delta = \min\{m, 10\}$ and $\gamma = 14$.

<table>
<thead>
<tr>
<th>$m$</th>
<th>Shift-Plus</th>
<th>($\delta, \gamma$)-Tuned-Boyer-Moore</th>
<th>($\delta, \gamma$)-Skip-Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>50.88</td>
<td>23.16</td>
<td>31.99</td>
</tr>
<tr>
<td>9</td>
<td>50.86</td>
<td>28.70</td>
<td>36.40</td>
</tr>
<tr>
<td>10</td>
<td>51.87</td>
<td>33.74</td>
<td>39.58</td>
</tr>
<tr>
<td>20</td>
<td>51.11</td>
<td>32.53</td>
<td>37.38</td>
</tr>
</tbody>
</table>

Table 7: Running times for ($\delta, \gamma$)-approximation with $\delta = \min\{m, 10\}$ and $\gamma = 15$.

<table>
<thead>
<tr>
<th>$m$</th>
<th>Shift-Plus</th>
<th>($\delta, \gamma$)-Tuned-Boyer-Moore</th>
<th>($\delta, \gamma$)-Skip-Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>50.72</td>
<td>23.33</td>
<td>32.02</td>
</tr>
<tr>
<td>9</td>
<td>50.70</td>
<td>27.96</td>
<td>35.65</td>
</tr>
<tr>
<td>10</td>
<td>51.94</td>
<td>33.88</td>
<td>40.00</td>
</tr>
<tr>
<td>20</td>
<td>51.35</td>
<td>33.20</td>
<td>37.03</td>
</tr>
</tbody>
</table>

Table 8: Running times for ($\delta, \gamma$)-approximation with $\delta = \min\{m, 10\}$ and $\gamma = 16$.

One should notice that the Shift-And and Shift-Plus algorithms need constant time to run whatever the values of the parameters are. In case of very high values for $\delta$ and/or $\gamma$ they have to be considered as the best choice.
<table>
<thead>
<tr>
<th>$m$</th>
<th>Shift-Plus</th>
<th>$(\delta, \gamma)$-Tuned-Boyer-Moore</th>
<th>$(\delta, \gamma)$-Skip-Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>50.67</td>
<td>23.29</td>
<td>32.20</td>
</tr>
<tr>
<td>9</td>
<td>50.83</td>
<td>28.38</td>
<td>35.74</td>
</tr>
<tr>
<td>10</td>
<td>51.93</td>
<td>34.41</td>
<td>39.91</td>
</tr>
<tr>
<td>20</td>
<td>50.18</td>
<td>32.94</td>
<td>37.10</td>
</tr>
</tbody>
</table>

Table 9: Running times for $(\delta, \gamma)$-approximation with $\delta = \min\{m, 10\}$ and $\gamma = 17$.  

<table>
<thead>
<tr>
<th>$m$</th>
<th>Shift-Plus</th>
<th>$(\delta, \gamma)$-Tuned-Boyer-Moore</th>
<th>$(\delta, \gamma)$-Skip-Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>51.24</td>
<td>23.57</td>
<td>32.22</td>
</tr>
<tr>
<td>9</td>
<td>50.31</td>
<td>28.33</td>
<td>35.73</td>
</tr>
<tr>
<td>10</td>
<td>51.83</td>
<td>34.36</td>
<td>40.15</td>
</tr>
<tr>
<td>20</td>
<td>49.97</td>
<td>32.77</td>
<td>37.03</td>
</tr>
</tbody>
</table>

Table 10: Running times for $(\delta, \gamma)$-approximation with $\delta = \min\{m, 10\}$ and $\gamma = 18$.

8 Conclusions

Here we presented the Skip-Search, Tuned-Boyer-Moore and Maximal-Shift approximate string matching algorithms that outperform the one presented in [4].

References


