Occurrence and substring heuristics for δ-matching

Maxime Crochemore, Costas S. Iliopoulos, Thierry Lecroq, Yoan J. Pinzon, Wojciech Plandowski, Wojciech Rytter

Fundamenta Informaticae XX (2003) 1–21
IOS Press

Occurrence and Substring Heuristics for δ-Matching

Maxime Crochemore[*]
Institut Gaspard Monge, Université Marne-la-Vallée, 77454 Marne-la-Vallée CEDEX 2, France and King's College London, UK, mac@univ-mlv.fr

Thierry Lecroq[*]
LIFAR–ABISS, Faculté des Sciences et Techniques, Université de Rouen, 76821 Mont-Saint-Aignan CEDEX, France Thierry.Lecroq@univ-rouen.fr

Wojciech Plandowski
Instytut Informatyki, Uniwersytet Warszawski, ul. Banacha 2, 02-097, Warszawa, Poland W.Plandowski@mimuw.edu.pl

Costas S. Iliopoulos[†*]
Dept. of Computer Science, King's College London, London WC2R 2LS, U.K., and School of Computing, Curtin University of Technology, GPO Box 1987 U, WA. Australia, csi@dcs.kcl.ac.uk

Yoan J. Pinzon[*]
Dept. of Computer Science, King's College London, London WC2R 2LS, U.K., pinzon@dcs.kcl.ac.uk

Wojciech Rytter
Instytut Informatyki, Uniwersytet Warszawski, ul. Banacha 2, 02-097, Warszawa, Poland, and Dept. of Computer Science, New Jersey Institute of Technology, USA, W.Rytter@mimuw.edu.pl

Abstract. We consider a version of pattern matching useful in processing large musical data: δ-matching, which consists in finding matches which are δ-approximate in the sense of the distance measured as maximum difference between symbols. The alphabet is an interval of integers, and the distance between two symbols a, b is measured as $|a-b|$. We also consider (δ, γ)-matching, where γ is a bound on the total sum of the differences. We first consider "occurrence heuristics" by adapting exact string matching algorithms to the two notions of approximate string matching. The resulting algorithms are efficient in practice. Then we consider "substring heuristics". We present δ-matching algorithms fast on the average providing that the pattern is "non-flat" and the alphabet interval is large. The pattern is "flat" if its structure does not vary substantially. The algorithms, named δ-BM1, δ-BM2 and δ-BM3 can be thought as members of the generalized Boyer-Moore family of algorithms. The algorithms are fast on average. This is the first paper on the subject, previously only "occurrence heuristics" have been considered. Our substring heuristics are much stronger and refer to larger parts of texts (not only to single positions). We use δ-versions of suffix tries and subword

[*]The work of these authors was partially supported by NATO grant PST.CLG.977017.
[†]The work of this author was partially supported by Welcome foundation, Royal Society and EPSRC grants.

graphs. Surprisingly, in the context of δ-matching subword graphs appear to be superior compared with compact suffix trees.

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 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 [8]). 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 applicable to molecular biology (see [11, 14, 16]) and in particular in DNA sequencing by hybridization ([17]), reconstruction of DNA sequences from known DNA fragments (see [19, 20]), in human organ and bone marrow transplantation as well as the determination of evolutionary trees among distinct species ([19]).

The approximate matching problem has been used for a variety of musical applications (see overviews in McGettrick [15]; Crawford et al [8]; Rolland et al [18]; Cambouropoulos et al [5]). 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 δ. This paper focuses in one special type of approximation that arise especially in musical information retrieval, i.e. δ-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 δ-approximate matching (and γ-approximate matching). In δ-approximate matching, equal-length patterns consisting of integers match if each corresponding integer differs by not more than δ- 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 (γ-approximate matching is described in the next section).

In [6], a number of efficient algorithms for δ-approximate matching, using "occurrence heuristics" was 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 [3, 22]. It is possible to adapt fast and practical exact string matching algorithms to these kind of approximations. In this paper we will present the adaptations of the TUNED-BOYER-MOORE [13], the SKIP-SEARCH algorithm [7] and the

MAXIMAL-SHIFT algorithm [21] and present some experiments to assert that these adaptations are faster than the algorithms using the bitwise technique.

Then we present three new algorithms, using "substring heuristics": δ-BM1, δ-BM2 and δ-BM3. They can be thought as members of the Boyer-Moore family of algorithms. The two first algorithms implement a heuristic based on a suitable generalization of the suffix trees data structure. The third algorithm uses a heuristic that considers fingerprints for selected substrings of the pattern and compares them with corresponding fingerprints of substrings of the text to be processed. The algorithms are fast on average. We provide experimental results and observations on the suitability of the heuristics. Our algorithms are particularly efficient for "non-flat" patterns over large alphabet intervals, and many patterns are of this kind.

The paper is organized as follows. In the next section we present some basic definitions for strings and background notions for approximate matching. In section 3 we present the adaptation of TUNED-BOYER-MOORE, SKIP-SEARCH and MAXIMAL-SHIFT to δ- and (δ,γ)-approximate string matching algorithms. In section 4 we present the data structures for our "substring heuristics", the three algorithms which use them and an average case analysis for two of them. In section 5 we present experimental results for all these algorithms. Finally in section 6 we present our conclusions.

2. Background and basic string definitions

A *string* is a sequence of zero or more symbols from an alphabet Σ; the string with zero symbols is denoted by ε. The set of all strings over the alphabet Σ is denoted by Σ^*. A string x of length m is represented by $x[1..m]$, where $x[i] \in \Sigma$ for $1 \le i \le m$. A string w is a *substring* of x if $x = uwv$ 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 $|u|+|w|$ 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 concatenations of k copies of x is denoted by x^k.

Let x be a string of length m. The integer p is said to be a *period* of x, if $x[i] = x[i+p]$ for all $1 \le i \le m-p$. *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 Σ be an alphabet of integers and δ an integer. Two symbols a, b of Σ are said to be δ-approximate, denoted $a \stackrel{\delta}{=} b$ if and only if

$$|a-b| \le \delta\,.$$

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

$$|x| = |y|, \text{ and } x[i] \stackrel{\delta}{=} y[i], \forall\, i \in \{1,\ldots,|x|\}\,. \tag{2.1}$$

For a given integer γ we say that two strings x, y are γ-approximate, denoted $x \stackrel{\gamma}{=} y$ if and only if

$$|x| = |y|, \text{ and } \sum_{i=1}^{|x|} |x[i]-y[i]| \le \gamma\,. \tag{2.2}$$

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

3. Occurrence heuristics

The problem of *δ-approximate pattern matching* is formally defined as follows: given a string $y = y[1..n]$ and a pattern $x = x[1..m]$ compute all positions j of y such that

$$x \stackrel{\delta}{=} y[j..j+m-1]\,.$$

A naive solution to this problem is to build an Aho-Corasick automaton (see [1]) of all strings that are δ-approximate to x and then use the automaton to process y. The time required to build the automaton is $O(\delta^m)$ since there are $2 \times \delta + 1$ different letters that can be used in each position, thus this method is of no practical use for large values of δ and m. In [6] 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. In this section we present direct adaptations of exact string matching algorithms to the notion of δ-approximate string matching. The new algorithms use only heuristics on single positions of the pattern. Section 3.1 presents the adaptation of the TUNED-BOYER-MOORE exact string matching algorithm. Section 3.2 depicts the adaptation of the SKIP-SEARCH exact string matching algorithm. Section 3.3 presents the adaptation of the MAXIMAL-SHIFT exact string matching algorithm. Section 3.4 shows the adaptation of these algorithms to (δ, γ)-approximate string matching.

3.1. δ-TUNED-BOYER-MOORE Approximate Pattern Matching

Here we present an adaptation of the TUNED-BOYER-MOORE for exact pattern matching algorithm to δ-approximate pattern matching. The exact pattern matching problem consists in finding one or more (generally all) exact occurrences of a pattern x of length m in a text y 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 [13] is a very fast practical variant of the famous BOYER-MOORE algorithm [4]. It only uses the occurrence shift function to perform the shifts. The occurrence shift function is defined for each symbol a in the alphabet Σ as follows:

$$\mathit{shift}[a] = \min\{\{m - i \mid x[i] = a \text{ with } 1 \leq i \leq m\} \cup \{m\}\}\,.$$

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 $x[m]$ 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 $x[m]$ in $x[1..m-1]$, if any. The length s of this shift is defined as follows:

$$s = \min\{\{m - i \mid x[i] = x[m] \text{ and } 0 < i < m\} \cup \{m\}\}\,.$$

To do δ-approximate pattern matching, the shift function can be defined to be for each symbol a in the alphabet Σ the distance from the right end of the pattern of the closest symbol $x[i]$ such that $x[i] \stackrel{\delta}{=} a$:

$$\mathit{shift}[a] = \min\{\{m - i \mid x[i] \stackrel{\delta}{=} a \text{ with } 1 \leq i \leq m\} \cup \{m\}\}\,.$$

```
δ-TUNED-BOYER-MOORE(x, m, y, n, δ)
 1  ▷ Preprocessing
 2  for all a ∈ Σ
 3       do shift[a] ← min({m − i | x[i] ≙δ a} ∪ {m})
 4  s ← min({m − i | x[i] ≙2δ x[m] and 0 < i < m} ∪ {m})
 5  y[n + 1..n + m] ← (x[m])^m
 6  ▷ Searching
 7  j ← m
 8  while j ≤ n
 9       do k ← shift[y[j]]
10          while k ≠ 0
11               do j ← j + k
12                  k ← shift[y[j]]
13                  j ← j + k
14                  k ← shift[y[j]]
15                  j ← j + k
16                  k ← shift[y[j]]
17          if x[1..m − 1] ≙δ y[j − m + 1..j − 1] and j ≤ n
18            then REPORT(j − m + 1)
19          j ← j + s
```

Figure 1. Adaptation of the TUNED-BOYER-MOORE exact pattern matching algorithm to do δ-approximate pattern matching. Line 3, m copies of $x[m]$ are appended at the end of y. Line 5, *shift*$[x[m]]$ is set to 0 so that during the inner loop (lines 10–16) of the searching phase whenever j becomes greater than n, k becomes equal to 0.

Then the length of the shift s becomes:

$$s = \min\{\{m - i \mid x[i] \overset{2\delta}{=} x[m] \text{ and } 0 < i < m\} \cup \{m\}\}\,.$$

The reason why it is necessary to use 2δ in the new definition of s is that 2δ is the minimum such that for any three symbols $a, b, c \in \Sigma$, if $a \overset{\delta}{=} b$ and $b \overset{\delta}{=} c$ then $a \overset{2\delta}{=} c$.

The pseudo-code for δ-TUNED-BOYER-MOORE algorithm can be found in figure 1.

3.2. δ-SKIP-SEARCH Approximate Pattern Matching

In the SKIP-SEARCH algorithm [7], for each symbol of the alphabet, a bucket collects all of that symbol's positions in x. When a symbol occurs k times in the pattern, there are k corresponding positions in the

```
δ-SKIP-SEARCH(x, m, y, n, δ)
 1  ▷ Preprocessing
 2  for all a ∈ Σ
 3       do z[a] ← {i | x[i] ≐^δ a}
 4  ▷ Searching
 5  j ← m
 6  while j ≤ n
 7       do for all i ∈ z[y[j]]
 8            do if x ≐^δ y[j − i + 1..j − i + m]
 9                 then REPORT(j − i + 1)
10          j ← j + m
```

Figure 2. Adaptation of the SKIP-SEARCH exact pattern matching algorithm to do δ-approximate pattern matching.

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:

$$z[a] = \{i \mid x[i] = a \text{ with } 1 \leq i \leq m\}\,.$$

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

To do δ-approximate pattern matching, the buckets can be computed as follows:

$$z[a] = \{i \mid x[i] \stackrel{\delta}{=} a \text{ with } 1 \leq i \leq m\}\,.$$

Figure 2 shows the pseudo-code for δ-SKIP-SEARCH algorithm. In this case when m is much shorter than the alphabet and the pattern x is flat (i.e. its structure does not vary substantially), many buckets are empty.

3.3. δ-MAXIMAL-SHIFT Approximate Pattern Matching

Sunday [21] 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

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

and a function *shift* such that

$$\mathit{shift}[\sigma(i)] \geq \mathit{shift}[\sigma(i+1)]$$

for $1 \leq i < m$ and

$$\mathit{shift}[\sigma(i)] = \min\{\ell \mid (\forall\, j \text{ such that } 1 \leq j < i, x[\sigma(j) - \ell] = x[\sigma(j)]) \text{ and } (x[\sigma(i) - \ell] \neq x[\sigma(i)])\}$$

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

We also defi ne a function bc for each symbol of the alphabet:

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

for $a \in \Sigma$.

Then, when the pattern is aligned with the $y[j..j+m-1]$ the comparisons are performed in the following order $x[\sigma(1)], x[\sigma(2)], \ldots, x[\sigma(m)]$ until the whole pattern is scanned or a mismatch is found. If a mismatch is found when comparing $x[\sigma(i)]$ then a shift of length $\max\{\mathit{shift}[\sigma(i)], bc[y[j+m+1]]\}$ 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 $bc[y[j+m+1]]$. Then the comparisons resume with $x[\sigma(1)]$ without keeping any memory of the comparisons previously done.

To perform δ-approximate string matching the two functions can be redefi ned as follows:

$$\mathit{shift}[\sigma(i)] = \min\{\ell \mid (\forall\, j \text{ such that } 1 \leq j < i, x[\sigma(j) - \ell] \stackrel{2\delta}{=} x[\sigma(j)]) \text{ and } (x[\sigma(i) - \ell] \stackrel{\delta}{\neq} x[\sigma(i)])\}$$

for $1 \leq i \leq m$ and

$$\mathit{shift}[m+1] = \min\{\ell \mid x[i] \stackrel{\delta}{=} x[i+\ell] \text{ for } 1 \leq i \leq m-\ell\}$$

and

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

for $a \in \Sigma$.

The preprocessing phase can be done in $O(m^2)$. Figure 3 gives the pseudo-code of the searching phase.

3.4. (δ, γ)-Approximate String Matching Algorithms

The problem of (δ, γ)-*approximate pattern matching* is formally defi ned as follows: given a string $y = y[1..n]$ and a pattern $x = x[1..m]$ compute all positions j of y such that

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

In [6] this problem was solved by making use of the SHIFT-AND algorithm to fi nd the δ-approximate matches of the pattern x in y. Once a δ-approximate match was found, it was then tested to check

```
δ-MAXIMAL-SHIFT(x, m, y, n, δ, shift, bc)
1  ▷ Searching
2  j ← 0
3  while j ≤ n − m
4      do i ← 1
5          while i ≤ m and x[σ(i)] = y[j + σ(i)]
6              do i ← i + 1
7          if i > m
8            then REPORT(j)
9          j ← j + max{shift[σ(i)], bc[y[j + m + 1]]}
```

Figure 3. Adaptation of the MAXIMAL-SHIFT exact pattern matching algorithm to do δ-approximate pattern matching.

whether it is also a γ-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 δ-TUNED-BOYER-MOORE, δ-SKIP-SEARCH and δ-MAXIMAL-SHIFT algorithms to the case of (δ, γ)-approximation, it just suffices to adapt the naive check of the pattern. The resulting algorithms are named (δ, γ)-TUNED-BOYER-MOORE algorithm, (δ, γ)-SKIP-SEARCH algorithm and (δ, γ)-MAXIMAL-SHIFT algorithm.

4. Substring heuristics

In this section we introduce "substring heuristics" to solve the problem of δ- and (δ, γ)-approximate string matching. It is the first time that such heuristics are considered for this kind of problems. Section 4.1 defines two notions of approximate dictionaries which corresponding data structures are given in sections 4.2 and 4.3. In section 4.4 we give three new algorithms using these approximate dictionaries and we give an average case analysis for two of them in section 4.5.

4.1. Two approximate dictionaries

The Boyer-Moore type algorithms are very efficient on average since scanning a small segment of size k allows, on average, to make large shifts of the pattern. Eventually this gives sublinear average time complexity. This general idea has many different implementations, see [9]. In this section, our approach to δ-matching is similar, we scan a segment of size k in the text. If this segment is not δ-approximate with any subword of the pattern we know that no occurrence of the pattern starts at $m - k$ positions to the left of the scanned segment. This allows to make a large shift of size $m - k$. The choice of k affects the complexity. In practice small k would suffice. Hence the first issue, with this approach, is to have a data structure which allows to check *fast* if a word of size k is δ-approximate to a subword of x. We are

especially interested in the answer "no" which allows to make a large shift, so an important parameter is the *rejection ratio*, denote by *Exact-RR*. It is the probability that a randomly chosen k-subword is not δ-approximate with a subword of x. If this ratio is high then our algorithms would work much faster on average. However another parameter is the time to check if the answer is "no". It should be proportional to k. We do a compromise: build a data structure with smaller rejection ratio but with faster queries about subwords of size k. Smaller rejection ratio means that sometimes we have answer "yes" though it should be "no", however if the real answer is "no" then our data structure always outputs "no" also. This is the negative answer which speeds up Boyer-Moore type algorithms. The positive answer has small effect. The data structure is an approximate one, its rejection ratio is denoted by *RR*, and it is hard to analyze it exactly. Hence we rather deal with heuristics. The performance depends on particular structure, the parameter k and class of patterns. Another important factors are rejection ratios: *Exact-RR* and *RR*. If *Exact-RR* is too small we cannot expect the algorithms to be very fast. On the other hand we need to have *RR* as close to *Exact-RR* as possible. The applicability is verified in practice. The starting structure is the suffix trie, it is effective in searching but it could be too large theoretically, though in practice k is small and k-truncated suffix trie is also small. Surprisingly we do not have linear size equivalent of (compact) suffix trees, but we have a linear size equivalent of subword graphs: δ-subword graphs. this shows that suffix trees and subword graphs are very different in the context of δ-matching. Below we give a formal definition of our data structures and rejection ratios. Denote by $SUB(x,k)$ the set of all substrings of x of size k. Denote also:

$$\delta\text{-}SUB(x,k) = \{z \mid z \stackrel{\delta}{=} w \text{ for some } w \in SUB(x,k)\}\,.$$

This is similar to the computation of high-scoring words in BLAST [2].

An *approximate dictionary* for a given string x is the data structure $\mathcal{D}_{x,\delta,k}$ which answers the queries:

$$\mathcal{D}_{x,\delta,k}(z) : \text{"}z \in \delta\text{-}SUB(x,k)\ ?\text{"}$$

Let $\mathcal{D}_{x,\delta,k}(z)$ be the result (*true* or *false*) of such query for a string z given by the data structure $\mathcal{D}_{x,\delta,k}$. By $\mathcal{D}_{x,\delta,\gamma,k}$ we denote the corresponding data structure for the queries involving the equality $z \stackrel{\delta,\gamma}{=} w$. In order for our data structure to work fast we allow that the answers could be incorrect. $\mathcal{D}_{x,\delta,k}(z)$ can answer *true* although z is not in δ-$SUB(x,k)$. By an efficiency of $\mathcal{D}_{x,\delta,k}$ we understand the *rejection-ratio* proportion:

$$RR_k(\mathcal{D}_{x,\delta,k}) = \frac{|\{z \in \Sigma^k \mid \mathcal{D}_{x,\delta,k}(z) = false\}|}{|\Sigma|^k}\,.$$

Optimal efficiency is the exact *rejection-ratio* for x:

$$\textit{Exact-}RR_k(x) = 1 - \frac{|\delta\text{-}SUB(x,k)|}{|\Sigma|^k}\,.$$

In other words the efficiency RR_k is the probability that a random substring z of length k is not accepted by $\mathcal{D}_{x,\delta,k}$ and the efficiency $\textit{Exact-}RR_k$ is the probability that a random substring z of length k is not an element of δ-$SUB(x,k)$. Our data structures $\mathcal{D}$ are *partially correct*:

$$\delta\text{-}SUB(x,k) \subseteq \{z \mid \mathcal{D}_{x,\delta,k}(z) = true\}\,.$$

4.2. δ-suffix tries and δ-subword graphs

Denote $i \ominus \delta = \max\{i - \delta, \min\{\Sigma\}\}$ and $i \oplus \delta = \min\{i + \delta, \max\{\Sigma\}\}$. We define δ-suffix tries and δ-subword graphs algorithmically. The δ-suffix trie of a pattern x is built as follows:

- build the trie $T = (V, E)$ recognizing all the suffixes of x where V is the set of nodes and $E \subseteq V \times \Sigma \times V$ is the set of edges of T;
- replace each edge $(p, a, q) \in E$ by $(p, [\max\{0, a - \delta\}, \min\{\max\{\Sigma\}, a + \delta\}], q)$;
- for all the nodes $v \in V$, if there are two edges $(v, [a, b], p), (v, [c, d], q) \in E$ such that $[a, b] \cap [c, d] \neq \emptyset$ then merge p and q into a single new node s and replace $(v, [a, b], p)$ and $(v, [c, d], q)$ by one edge $(v, [\min\{a, c\}, \max\{b, d\}], s)$.

We have an equivalence relation on the set of vertices: two vertices are equivalent iff they are roots of isomorphic subtrees. In the δ-suffix trie construction we process nodes by taking at each *large* step all vertices which are in a same equivalence class C. Then in this step we process all edges outgoing from vertices from C. All these vertices are independent and we can think that it is done in parallel. The construction terminates when the trie stabilizes. The δ-subword graph of a sequence x is obtained by minimizing its δ-suffix trie. This means that each equivalence class of vertices is merged into a single vertex. Figure 5 shows an example of δ-suffix trie and δ-subword graph. It should be noted that using an alphabet of ranges is not a new idea (see [12]).

Theorem 4.1. The numbers of nodes and of edges of δ-subword graph for the string x are $O(|x|)$.

Proof:
The number of equivalence classes of the initial suffix trie is at most $2n$. In the process of merging edges the nodes which are equivalent initially will remain equivalent until the end. Hence the number of equivalence classes of intermediate δ-suffix trie (after processing all edges outgoing from nodes in a same equivalence class) is at most $2n$, which gives the upper bound on the number of nodes of the δ-subword graph. The bound on the number of edges can be shown similarly as for standard subword graphs. □

4.3. Families of intervals

For each subword $w \in SUB(x, k)$ of x, denote by $hash(w)$ the sum of the symbols of y. For each $k < |x|$ we introduce the following families of intervals (overlapping and adjacent intervals are "glued together") of the interval $[\min\{\Sigma\},\ k \times \max\{\Sigma\}]$ which represents respectively the sets:

$$\mathcal{F}_\delta(x, k) = \bigcup_{w \in SUB(x,k)} [hash(w) \ominus k\delta,\ hash(w) \oplus k\delta]$$

and

$$\mathcal{M}_\gamma(x, k) = \bigcup_{w \in SUB(x,k)} [hash(w) \ominus \min\{k\delta, \gamma\},\ hash(w) \oplus \min\{k\delta, \gamma\}]\,.$$

Clearly $\mathcal{M}_{k\delta}(x, k) = \mathcal{F}_\delta(x, k)$. Figure 6 presents an example.

The definitions of $\mathcal{F}$ and $\mathcal{M}$ imply:

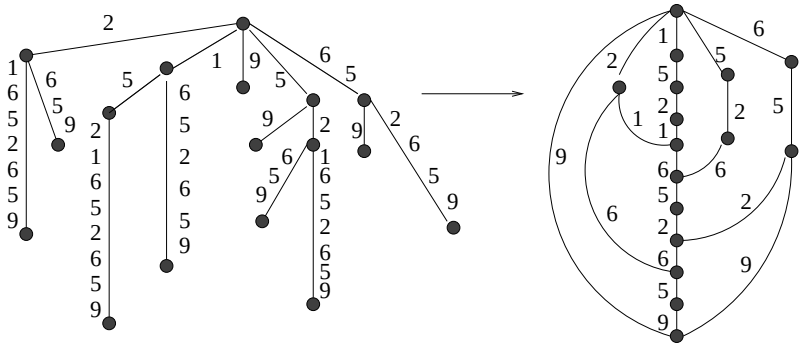

Figure 4. The suffix tree and subword graph for the word $w = 1521652659$ and $\Sigma = [0..9]$.

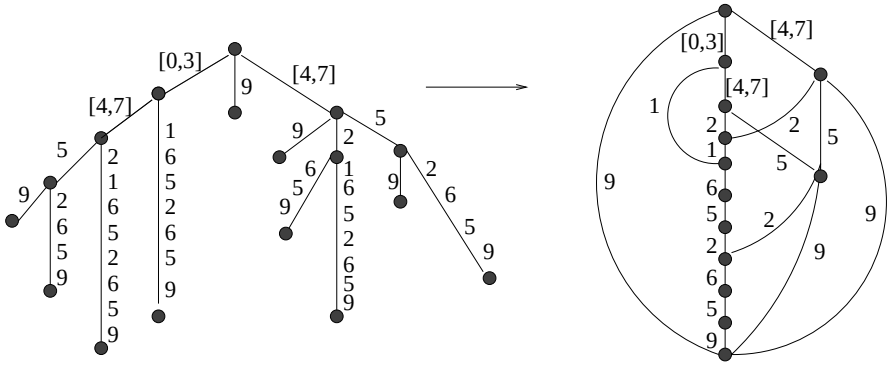

Figure 5. The δ-suffix tree and the δ-subword graph for the sequence $w = 1521652659$ with $\delta = 1$ and $\Sigma = [0..9]$. A single integer i means the interval $[i \ominus \delta, i \oplus \delta]$.

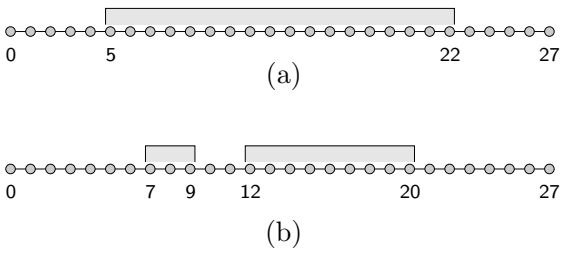

Figure 6. (a) The family of intervals $\mathcal{F}_\delta(x, k)$ and (b) the family $\mathcal{M}_\gamma(x, k)$, for the string 1529283 with $\delta = \gamma = 1$, $k = 3$ and $\Sigma = [0..9]$.

```
δ-BM1(x, m, y, n, δ)
1  i ← m
2  while i ≤ n
3       do if y[i − k + 1...i] ∈ δ-suffi x trie of x
4            then NAIVE(i, i + m − k − 1)
```

Figure 7. δ-BM1 algorithm.

Lemma 4.1. The two following properties hold:

(a) If $z \stackrel{\delta}{=} w$ for some $w \in SUB(x,k)$ then $hash(z) \in \mathcal{F}_\delta(x,k)$;

(b) If $z \stackrel{\delta,\gamma}{=} w$ for some $w \in SUB(x,k)$ then $hash(z) \in \mathcal{M}_\gamma(x,k)$.

The effi ciency of the family $\mathcal{I}$ of intervals can be measured as $RR(\mathcal{I}) = 1 - Prob(hash(z) \in \mathcal{I})$ where z is a random string of length k. In other words it is the probability that an integer is not in any interval of the family. Observe that $\mathcal{I}$ in our case is always represented as a family of disjoint intervals, overlapping and adjacent ones have been glued together.

4.4. Three δ-BM algorithms

We show how the data structures introduced in this section are used in δ-matching. We now want to fi nd all the δ-occurrences of a pattern x of length m in a text y of length n. We apply a very simple greedy strategy: place the pattern over the text such that the right end of the pattern is over position i in the text. Then check if the suffi x suf of length k (k may depend on x) of text ending at i is "sensible". If not the pattern is shifted by a large amount and many positions of of the text are never inspected at all. If suf is sensible then a naive search in a "window" on the text is performed. Figure 7 implements this method.

We denote here by NAIVE(p,q) a procedure checking directly if x ends at positions in the interval $[p..q]$, for $p < q$.

We design an improved version of δ-BM1 using δ-subword graphs instead of tries. The δ-subword graph of the reverse pattern is denoted by (Σ, V, v_0, F, E), where Σ is the alphabet, V is the set of states, $v_0 \in V$ is the initial state, $F \subseteq V$ is the set of fi nal states and $E \subseteq V \times \Sigma \times V$ is the set of transitions. Let $\delta\text{-}per(x)$ be the δ-period of the word x defi ned by $\delta\text{-}per(x) = \min\{p \mid \forall 1 \le i \le m-p, x[i] \stackrel{\delta}{=} x[i+p]\}$.

Then it is possible to adopt the same strategy as the Reverse Factor algorithm [9] for exact string matching to δ-approximate string matching. When the pattern x is compared with $y[i-m+1..i]$ the symbols of $y[i-m+1..i]$ are parsed through the δ-subword graph of the reverse pattern from right to left starting with the initial state. If transitions are defi ned for every symbol of $y[i-m+1..i]$, it means that a δ-occurrence of the pattern could have been found and the pattern can be shifted by $\delta\text{-}per(x)$ positions to the right. Otherwise the pattern can be shifted by m minus the length of the path, in the δ-subword graph, from the initial state and the last fi nal state encountered while scanning $y[i-m+1..i]$ from right to left. Indeed the δ-subword graph of the reverse pattern recognizes at least all the δ-suffi xes of the

```
δ-BM2(x, m, y, n, δ)
 1  i ← m
 2  while i ≤ n
 3      do q ← v_0
 4         j ← i
 5         b ← 0
 6         while (q, y[j], p) ∈ E
 7             do q ← p
 8                j ← j − 1
 9                if q ∈ F
10                  then b ← i − j
11         if i − j > m
12           then check and report a δ-occurrence at position i − m + 1
13                i ← i + δ-per(x)
14           else i ← i + m − b
```

Figure 8. δ-BM2 algorithm.

reverse pattern from right to left and thus at least all the δ-prefi xes of the pattern from left to right. Figure 8 implements this method.

The value $\delta\text{-}per(x)$ can be approximated using the δ-subword graph of the reverse pattern.

Our last algorithm can be used also for (δ, γ)-approximate string matching. We apply the data structure of interval families. Figure 9 implements this method. δ-BM3 algorithm is conceptually simpler than the other algorithms and its preprocessing is easy.

4.5. Average time analysis of algorithms δ-BM1 and δ-BM3

Denote $p = Prob(x \stackrel{\delta}{=} y)$ where x and y are random symbols and $q_{k,x} = RR_k(\mathcal{D}_{x,\delta,k})$.

Lemma 4.2. The overall average number of comparisons made by δ-BM1 and δ-BM3 algorithms is at most

$$\frac{n}{m-k}\left(k + (1-q_{k,x})\,\frac{m}{1-p}\right)$$

Proof:
Divide the text into windows of size $m-k$. In each window the probability that the pattern is moved to the next window after at most k comparisons is $q_{k,x}$. Now it is enough to prove that the average number of comparisons made by the naive algorithm in the window is bounded by $\frac{m}{1-p}$. We assume now that our algorithm performs worse than in reality and no matter what is the result of comparisons of symbols at positions $m-k+1..m$ of the window the algorithm goes further and ends when one of the other window

```
δ-BM3(x, m, y, n, δ)
1  i ← m
2  while i ≤ n
3       do if hash(y[i − k + 1..i]) ∈ M_{δ,γ}(x, k)
4            then NAIVE(i, i + m − k − 1)
```

Figure 9. δ-BM3 algorithm.

symbols mismatches the symbol of the pattern or all of them match. This is because we cannot assume that the symbols at positions $m-k+1..m$ are random since they matched the symbols in the dictionary.

Making this assumption the average number of comparisons made by the naive algorithm at position m of the window is

$$p_1 = (1-p)(k+1) + p(1-p)(k+2) + p^2(1-p)(k+3) + \cdots +$$

$$+p^{m-k-1}(1-p)m + p^{m-k}m.$$

Similarly the average number of comparisons made by the naive algorithm at position $m+1$ is

$$p_2 = (1-p)\cdot 1 + p(1-p)(k+2) + p^2(1-p)(k+3) + \cdots + p^{m-k-1}(1-p)m + p^{m-k}m.$$

Similarly the average number of comparisons made by the naive algorithm at position $m+j-1$ for $j = 1..m-k$ is

$$p_j = (1-p)\cdot 1 + p(1-p)\cdot 2 + \cdots + p^{j-2}(1-p)\cdot(j-1) +$$

$$+p^{j-1}(1-p)(k+j) + p^j(1-p)(k+j+1) + \ldots p^{m-k-1}(1-p)m + p^{m-k}m.$$

We have

$$p_j = (1-p)\cdot 1 + p(1-p)\cdot 2 + \cdots + p^{m-k-1}(1-p)(m-k) +$$

$$+k(p^{j-1}(1-p) + p^j(1-p) + \cdots + p^{m-k-1}(1-p)) + p^{m-k}m =$$

$$\frac{1-p^{m-k}}{1-p} - (m-k)p^{m-k} + k(p^{j-1} - p^{m-k}) + p^{m-k}m =$$

$$\frac{1-p^{m-k}}{1-p} + kp^{j-1}.$$

Hence,

$$\sum_{j=1}^{m-k} p_j = (m-k)\frac{1-p^{m-k}}{1-p} + k\frac{1-p^{m-k}}{1-p} = m\frac{1-p^{m-k}}{1-p} \leq \frac{m}{1-p}.$$

This completes the proof. □

The analysis of our algorithms would be simpler and we would get slightly better estimation for the average number of symbol comparisons if we assume that in each window the naive algorithm performs comparisons of text symbols at positions $m-k+1..m$ at the end. The estimation is however not significantly better. In particular it does not give an improvement in the estimation in our next theorem.

Theorem 4.2. Let $k \leq 0.99m$ and $p \leq 0.99$. The average time complexity of the algorithms δ-BM1 and δ-BM3 is

$$O(\frac{n}{m}(k+(1-q_{k,x})m)) .$$

Observe here that our analysis does not depend on the data structure $\mathcal{D}$. The only thing it assumes is that the scheme of the algorithm matches the structure of the algorithms δ-BM1 and δ-BM3. Clearly, the efficiency of such algorithms depends heavily on the choice of k and the efficiency of $\mathcal{D}$. For instance, for $\delta = 0$, (ie. we consider string matching without errors) we may choose $k = 2\log_{|\Sigma|} m$. Then, for δ-BM1, $1-q_{k,x}$ is the probability that a random string of length k is not a subword of x. The number of subwords of length k of x is at most m and the number of all words of length k is m^2 so $1-q_{k,x} \leq \frac{1}{m}$ thus the average time complexity is $O(\frac{n}{m}\log m)$, the best possible. Moreover k may depend also on the pattern x itself. If p is "good" then k may be chosen small and when it is "bad" k may be chosen bigger. In particular we may increase k up to the moment when $1-q_{k,x}$ decreases below an acceptable level.

5. Experimental Results

We first count the number of text character inspections of the algorithms in section 5.1 while we observe their running times in section 5.2.

5.1. Text character inspections

We computed experimentally the values *RR* and *Exact-RR* for our approximate dictionaries for various values of k and different sizes of the alphabet. These efficiencies correspond to average case complexity of our δ-BM algorithms. We compared the values of *RR* and *Exact-RR* with average running time for sufficiently large sample of random inputs. We counted the average number of text character inspections for the following algorithms: δ-TUNED-BOYER-MOORE, δ-SKIP-SEARCH, δ-MAXIMAL-SHIFT [10] and δ-BM1, δ-BM2 and δ-BM3.

All the algorithms have been implemented in C in a homogeneous way such as to keep their comparison significant. The text used is composed of $500,000$ symbols and was randomly built. The size of the alphabet is 100. The target machine is a PC, with a AMD-K6 II processor at 330MHz running Linux kernel 2.2. The compiler is `gcc`. For each pattern length m, we searched per one hundred patterns randomly built.

We counted the number c of text character inspections for one text character. The results are presented in figures 10 and 11. For $\delta = 1$ the best results for δ-BM1 algorithm have been obtained with $k = \log_2 m$. The best results for the δ-BM3 algorithm have always been obtained with $k = 2$. For small values of δ, δ-BM1 and δ-BM2 algorithms are better than δ-TUNED-BOYER-MOORE algorithm (which is the best among the known algorithms) for large values of m ($m \geq 20$). For large values of m, δ-BM1 and δ-BM2 algorithms are performing a large number of text character inspections since the δ-subword graph and the δ-suffix trie tend to be a line with all edges labeled with $[\min\{\Sigma\}..\max\{\Sigma\}]$. For larger

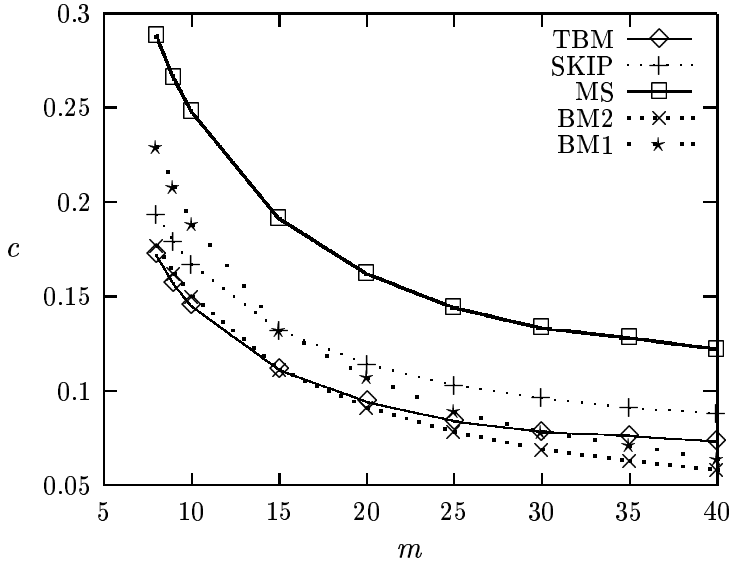

Figure 10. Results for $\delta = 1$.

values of δ (up to 5) δ-BM1 and δ-BM2 algorithms are better than δ-TUNED-BOYER-MOORE algorithm for small values of m ($m \leq 12$). For larger values of δ, the δ-TUNED-BOYER-MOORE algorithm is performing better than the other algorithms. In conclusion the algorithms introduced in this article are of particular practical interest for large alphabets, short patterns and small values of δ. Alphabets used for music representations are typically very large. A "bare" absolute pitch representation can be base-7 (7 symbols), base-12, base-40 or 120 symbols for MIDI. But meaningful alphabets that will allow us to do in-depth music analysis use symbols that in reality is a set of parameters. A typical symbol could be $(a_1, a_2, a_3, \ldots, a_k)$, where a_1 represents the pitch, a_2 represents the duration, a_3 the accent etc. A typical pattern ("motif") in musical sequence is 15-20 notes but an alphabet can have thousands of symbols. Thus the need of algorithms that perform well for small patterns and large alphabets.

5.2. Running times

We implemented in C, in a homogeneous way, the following algorithms:

SHIFT-AND, δ-TUNED-BOYER-MOORE, δ-SKIP-SEARCH, δ-MAXIMAL-SHIFT and

SHIFT-PLUS, (δ, γ)-TUNED-BOYER-MOORE, (δ, γ)-SKIP-SEARCH and (δ, γ)-MAXIMAL-SHIFT.

We randomly built a text of $500,000$ symbols on an alphabet of size 70. We then searched for each values of m, 100 random patterns and took the average running time. Times are measured in hundredth of seconds and include both preprocessing and searching times. The design of efficient algorithms for the preprocessing phase of the algorithms using "substring heuristics" is still an open problem, they are not included in the running time experimentation.

The results for δ-approximation are shown in tables 1 to 5. For the values used in these experiments, the δ-TUNED-BOYER-MOORE algorithm is always faster than the δ-SKIP-SEARCH algorithm which is itself always faster than the SHIFT-AND algorithm.

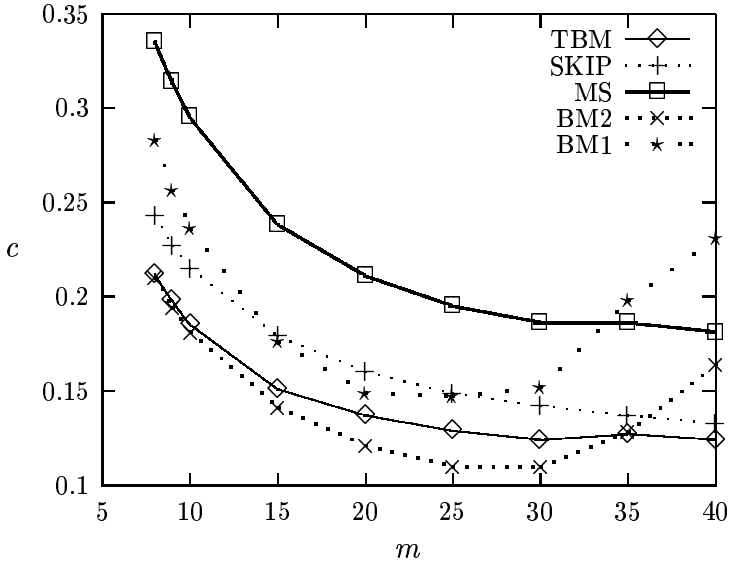

Figure 11. Results for $\delta = 2$.

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

m	SHIFT-AND	δ-TUNED-BOYER-MOORE	δ-SKIP-SEARCH
8	32.98	10.78	18.61
9	32.90	10.55	18.11
10	32.93	10.10	17.65
20	32.86	9.32	15.81

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

m	SHIFT-AND	δ-TUNED-BOYER-MOORE	δ-SKIP-SEARCH
8	33.07	13.40	21.66
9	32.90	13.00	20.94
10	32.93	12.64	20.49
20	32.92	11.97	18.81

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

m	SHIFT-AND	δ-TUNED-BOYER-MOORE	δ-SKIP-SEARCH
8	33.65	16.65	24.99
9	33.14	16.05	24.06
10	33.05	15.71	23.62
20	32.93	14.82	21.42

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

m	SHIFT-AND	δ-TUNED-BOYER-MOORE	δ-SKIP-SEARCH
8	34.72	21.18	29.15
9	33.41	20.03	27.64
10	33.07	19.12	26.85
20	32.81	18.20	24.41

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

m	SHIFT-AND	δ-TUNED-BOYER-MOORE	δ-SKIP-SEARCH
8	36.46	26.82	34.64
9	34.46	24.36	31.46
10	33.41	23.61	30.55
20	33.00	22.32	27.54

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

m	SHIFT-PLUS	(δ, γ)-TUNED-BOYER-MOORE	(δ, γ)-SKIP-SEARCH
8	50.73	23.33	31.93
9	50.32	27.78	35.52
10	51.79	33.76	39.45
20	50.26	32.46	36.91

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

m	SHIFT-PLUS	(δ, γ)-TUNED-BOYER-MOORE	(δ, γ)-SKIP-SEARCH
8	50.88	23.16	31.99
9	50.86	28.70	36.40
10	51.87	33.74	39.58
20	51.11	32.53	37.38

The results for (δ, γ)-approximation are shown in tables 6 to 10. For the values that were used in these experiments, the (δ, γ)-TUNED-BOYER-MOORE algorithm is always faster than the (δ, γ)-SKIP-SEARCH algorithm which is itself always faster than the SHIFT-PLUS algorithm.

Experiments conduct only on γ-approximation show that an adaptation to this case of the SKIP-SEARCH algorithm is faster than an adaptation of the TUNED-BOYER-MOORE algorithm.

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 δ and/or γ they have to be considered as the best choice.

6. Conclusion

We presented in this article two types of heuristics for δ- and γ-string matching problems. We first consider "occurrence heuristics" for which we designed the SKIP-SEARCH, TUNED-BOYER-MOORE and MAXIMAL-SHIFT approximate string matching algorithms that outperform, in practice, the one

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

m	SHIFT-PLUS	(δ, γ)-TUNED-BOYER-MOORE	(δ, γ)-SKIP-SEARCH
8	50.72	23.33	32.02
9	50.70	27.96	35.65
10	51.94	33.88	40.00
20	51.35	33.20	37.03

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

m	SHIFT-PLUS	(δ, γ)-TUNED-BOYER-MOORE	(δ, γ)-SKIP-SEARCH
8	50.67	23.29	32.20
9	50.83	28.38	35.74
10	51.93	34.41	39.91
20	50.18	32.94	37.10

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

m	SHIFT-PLUS	(δ, γ)-TUNED-BOYER-MOORE	(δ, γ)-SKIP-SEARCH
8	51.24	23.57	32.22
9	50.31	28.33	35.73
10	51.83	34.36	40.15
20	49.97	32.77	37.03

presented in [6]. Then we consider "substring heuristics". Using δ-tries, δ-subword graphs and families of intervals we gave three new algorithms that can be considered as members of the Boyer-Moore family of pattern matching algorithms. The algorithms using "occurrence heuristics" are very fast in practice. A theoretical study of the algorithms using "substring heuristics" shows that they are optimal in the average. An experimental study shows that for small values of δ (1 or 2) the δ-BM algorithms are efficient. For mid values of δ the TUNED-BOYER-MOORE algorithm is the fastest algorithm. In case of very high values for δ and/or γ the SHIFT-AND and SHIFT-PLUS algorithms have to be considered as the best choice.

A problem remains in designing efficient algorithms for building δ-tries and δ-subword graphs.

Acknowledgements

The authors are very grateful to the anonymous referees who greatly improved the quality of this article.

References

[1] Aho, A. V., Corasick, M. J.: Efficient string matching: an aid to bibliographic search, *Communications of the ACM*, **18**(6), 1975, 333–340.

[2] Altschul, S. F., Gish, W., Miller, W., Myers, E. W., Lipman, D. J., Basic local alignment search tool, *Journal of Molecular Biology*, **215**(3), 1990, 403–410.

[3] Baeza-Yates, R., Gonnet, G.: A new approach to text searching, *Communications of the ACM*, **35**(10), 1992, 74–82.

[4] Boyer, R. S., Moore, J. S.: A fast string searching algorithm, *Communications of the ACM*, **20**(10), 1977, 762–772.

[5] Cambouropoulos, E., Crawford, T., Iliopoulos, C. S.: Pattern Processing in Melodic Sequences: Challenges, Caveats and Prospects, *Proc. of the AISB'99 Convention (Artifi cial Intelligence and Simulation of Behaviour)*, Edinburgh, U.K., 1999, 42–47.

[6] Cambouropoulos, E., Crochemore, M., Iliopoulos, C. S., Mouchard, L., Pinzon, Y. J.: Algorithms for computing approximate repetitions in musical sequences, *Proc. 10th Australasian Workshop On Combinatorial Algorithms* (R. Raman, J. Simpson, Eds.), Perth, WA, Australia, 1999, 129–144.

[7] Charras, C., Lecroq, T., Pehoushek, J. D.: A very fast string matching algorithm for small alphabets and long patterns, *Proc. 9th Annual Symposium on Combinatorial Pattern Matching* (M. Farach-Colton, Ed.), Piscataway, NJ, LNCS 1448, Springer-Verlag, Berlin, 1998, 55–64.

[8] Crawford, T., Iliopoulos, C. S., Raman, R.: String Matching Techniques for Musical Similarity and Melodic Recognition, *Computing in Musicology*, **11**, 1998, 73–100.

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

[10] Crochemore, M., Iliopoulos, C. S., Lecroq, T., Pinzon, Y. J.: Approximate string matching in musical sequences, *Proc. Prague Stringology Conference'01* (M. Balík, M. Simánek, Eds.), Prague, Tcheque Republic, 2001, Annual Report DC–2001–06, 26–36.

[11] Fischetti, V. A., Landau, G. M., Schmidt, J. P., Sellers, P. H.: Identifying periodic occurrences of a template with applications to protein structure, *Proc. 3rd Annual Symposium on Combinatorial Pattern Matching*, LNCS 644, 1992, 111–120.

[12] Fredkin, E.: Trie memory, *Communications of the ACM*, **3**(9), 1960, 490–499.

[13] Hume, A., Sunday, D. M.: Fast string searching, *Software–Practice and Experience*, **21**(11), 1991, 1221–1248.

[14] Karlin, S., Morris, M., Ghandour, G., Leung, M.-Y.: Efficient algorithms for molecular sequences analysis, *Proc. Natl. Acad. Sci. USA*, **85**(3), 1988, 841–845.

[15] McGettrick, P.: MIDIMatch: Musical Pattern Matching in Real Time, MSc Dissertation, York University, U.K., 1997.

[16] Milosavljevic, A., Jurka, J.: Discovering simple DNA sequences by the algorithmic significance method, *Computer Applications in Biosciences*, **9**(4), 1993, 407–411.

[17] Pevzner, P. A., Feldman, W.: Gray Code Masks for DNA Sequencing by Hybridization, *Genomics*, **23**, 1993, 233–235.

[18] Rolland, P.-Y., Ganascia, J.-G.: Musical Pattern Extraction and Similarity Assessment, *Readings in Music and Artifi cial Intelligence* (E. R. Miranda, Ed.), Harwood Academic Publishers, 2000, 115–144.

[19] Schmidt, J. P.: All shortest paths in weighted grid graphs and its application to finding all approximate repeats in strings, *SIAM Journal on Computing*, **27**(4), 1998, 972–992.

[20] Skiena, S. S., Sundaram, G.: Reconstructing strings from substrings, *Journal of Computational Biology*, **2**, 1995, 333–353.

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

[22] Wu, S., Manber, U.: Fast text searching allowing errors, *Communications of the ACM*, **35**(10), 1992, 83–91.