Three Heuristics for δ-Matching: δ-BM Algorithms

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

Three Heuristics for δ-Matching: δ-BM Algorithms

Maxime Crochemore[1,⋆], Costas S. Iliopoulos[2], Thierry Lecroq[3], Wojciech Plandowski[4], and Wojciech Rytter[5]

[1] Institut Gaspard Monge, Université Marne-la-Vallée, Cité Descartes, 5 Bd Descartes, Champs-Sur-Marne, 77454 Marne-la-Vallée CEDEX 2, France, mac@univ-mlv.fr, URL: http://www-igm.univ-mlv.fr/~mac

[2] 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, URL: http://www.dcs.kcl.ac.uk/staff/csi

[3] LIFAR–ABISS, Faculté des Sciences et Techniques, Université de Rouen, 76821 Mont-Saint-Aignan CEDEX, France, Thierry.Lecroq@univ-rouen.fr, URL: http://www-igm.univ-mlv.fr/~lecroq

[4] Instytut Informatyki, Uniwersytet Warszawski, ul. Banacha 2, 02-097, Warszawa, Poland, W.Plandowski@mimuw.edu.pl, URL: http://www.mimuw.edu.pl/~wojtekpl

[5] Instytut Informatyki, Uniwersytet Warszawski, ul. Banacha 2, 02-097, Warszawa, Poland, and Department of Computer Science, Liverpool University, Peach Street, Liverpool L69 7ZF, U.K., W.Rytter@mimuw.edu.pl, URL: http://www.mimuw.edu.pl/~rytter

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 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. We also consider (δ,γ)-matching, where γ is a bound on the total number of errors. 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 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 graphs. Surprisingly, in the context of δ-matching subword graphs appear to be superior compared with compacted suffix trees.

1 Introduction

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

⋆ The work of the three first authors was partially supported by NATO grant PST.CLG.977017.

set of intervals that appear between notes (e.g. pitch may be represented as MIDI numbers and pitch intervals as number of semitones). Approximate matching 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" [3]. Such algorithms can be particularly useful for melody identification and musical retrieval. The approximate matching problem has been used for a variety of musical applications (see overviews in [8, 3, 11, 1]). 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 amount of error 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 δ-approximate matching. In δ-approximate matching, equal-length patterns match if each corresponding integer differs by not more than δ. For example, a C-major $\{60, 64, 65, 67\}$ and a C-minor $\{60, 63, 65, 67\}$ sequence can be matched if a tolerance of $\delta = 1$ is allowed in the matching process. For a given sequence of length m, the total amount of error is bounded by $O(\delta \cdot m)$. This increases dramatically for large values of δ, and therefore, we can restrict it to a maximum of γ. This further restriction will be referred as (δ, γ)-approximate matching. In [2], a number of efficient algorithms for δ-approximate and (δ, γ)-approximate matching were presented (i.e. the SHIFT-AND algorithm and the SHIFT-PLUS algorithm, respectively). These algorithms use the bit-wise technique. In [5] exact string matching algorithms are adapted to δ-approximation using heuristics on the alphabet symbols. Here, we present three new algorithms: 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.

2 δ-approximate dictionaries

Let Σ be an alphabet of integers and δ an integer. Two symbols a, b of Σ are said to be δ-approximate, denoted $a \overset{\delta}{=} b$, iff $|a - b| \leq \delta$. We say that two

strings x, y are δ-approximate, denoted $x \stackrel{\delta}{=} y$ if and only iff $|x| = |y|$, and $x[i] \stackrel{\delta}{=} y[i]$, for each $i \in \{1..|x|\}$. For a given integer γ we say that two strings x, y are γ-approximate, denoted $x \stackrel{\gamma}{=} y$ if and only if

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

Two strings x, y are (δ, γ)-approximate, denoted $x \stackrel{\delta,\gamma}{=} y$, if and only if x and y satisfy both conditions above. 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 [4]. 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 *pat*. 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 *pat*. 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 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 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:

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

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

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

Let $\mathcal{D}_x(z)$ be the result (*true* or *false*) of such query for a string z given by the data structure $\mathcal{D}_x$. By $\mathcal{D}_x^1$ we denote the corresponding data structure for the queries involving the equality $z \stackrel{\delta,\gamma}{=} y$. In order for our data structure to work fast we allow that the answers could be incorrect. By an efficiency of $\mathcal{D}_x$ we understand the *rejection-ratio* proportion:

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

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

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

In other words the efficiency is the probability that a random substring z of length k is not accepted by $\mathcal{D}_x$ or (in the case of *Exact-RR*) it is not an element of δ-$SUB(x,k)$. Our data structures $\mathcal{D}$ are *partially correct*:

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

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 2 shows an example of δ-*suffix trie* and δ-subword graph.

Theorem 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. □

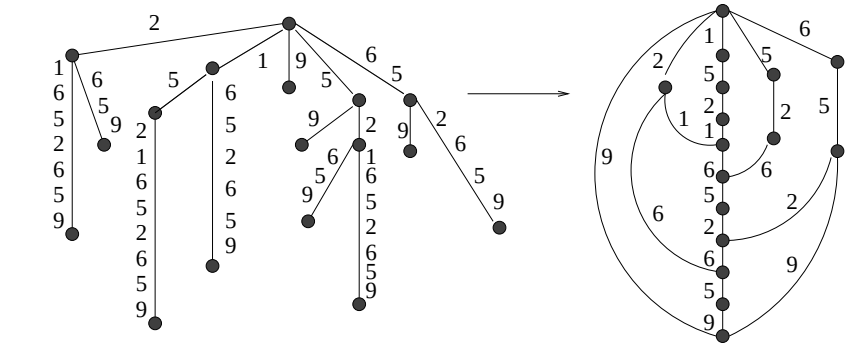

Fig. 1. The suffix trie and subword graph for the word $w = 1521652659$.

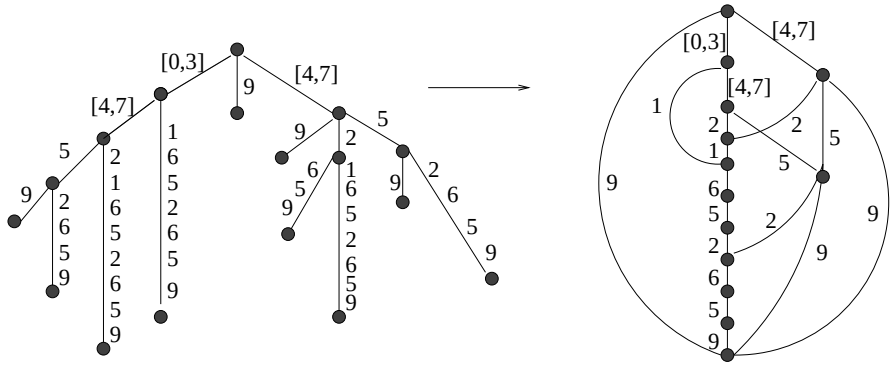

Fig. 2. The δ-*suffix trie* and the δ-subword graph for the sequence $w = 1521652659$ with $\delta = 1$ and $\Sigma = [0..9]$. A single integer means i means the interval $[i \ominus \delta, i \oplus \delta]$.

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

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

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

Clearly $\mathcal{M}_{k\delta}(x,k) = \mathcal{F}_\delta(x,k)$. Figure 3 presents an example.

Lemma 1.
(a) *If* $z \overset{\delta}{=} y$ *for some* $y \in SUB(x,k)$ *then* $hash(z) \in \mathcal{F}_\delta(x,k)$.
(b) *If* $z \overset{\delta,\gamma}{=} y$ *for some* $y \in SUB(x,k)$ *then* $hash(z) \in \mathcal{M}_\gamma(x,k)$.

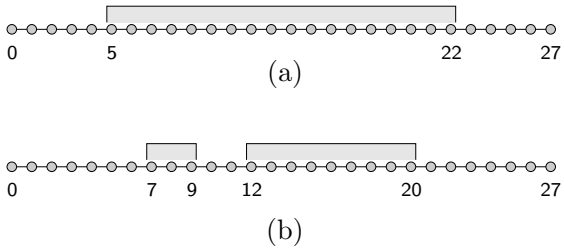

Fig. 3. (a) The family of intervals $\mathcal{F}_\delta(x,k)$ and (b) the family $\mathcal{M}_\gamma(x,k)$, for the string 1 5 2 9 2 8 3 with $\delta = \gamma = 1$, $k = 3$ and $\Sigma = [0..9]$.

The efficiency of the family $\mathcal{I}$ of intervals can be measured as $RR(\mathcal{I}) = 1 - Prob(hash(x) \in \mathcal{I})$ where x 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 adjoined ones have been glued together.

3 Three δ-BM algorithms

We show in this section how the data structures of Section 2 are used in δ-matching. We now want to find all the δ-occurrences of a pattern *pat* of length m in a text *text* 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 suffix *suf* of length k (k may depend on *pat*) of text ending at i is "sensible". If not the pattern is shifted by a large amount and many position of text are never inspected at all. If *suf* is sensible then a naive search in a "window" of text is performed.

```
Algorithm δ-BM1;
    i ← m;
    while i ≤ n
        if text[i − k + 1...i] ∈ δ-Suffix-Trie(pat)
            then NAIVE(i, i + m − k − 1);
        i ← i + m − k;
```

We denote here by $NAIVE(p,q)$ a procedure checking directly if *pat* ends at positions in the interval $[p..q]$, for $p < q$.
We design an improved version of δ-BM1 using δ-subword graphs instead tries. Let (Σ, V, v_0, F, E) be the δ-subword graph of the reverse pattern, 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 final 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 defined by

$$\delta\text{-}per(x) = \min\{p \mid \forall 1 \leq i \leq m-p \;\; x[i] \stackrel{\delta}{=} x[i+p]\}.$$

Then it is possible to adopt the same strategy as the Reverse Factor algorithm [4] for exact string matching to δ-approximate string matching. When the pattern pat is compared with $text[i-m+1 \ldots i]$ the symbols of $text[i-m+1 \ldots i]$ are parsed through the δ-subword graph of the reverse pattern from right to left starting with the initial state. If transitions are defined for every symbol of $text[i-m+1 \ldots i]$, it means that a δ-occurrence of the pattern has been found and the pattern can be shifted by δ-$per(pat)$ 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 final state encountered while scanning $text[i-m+1 \ldots i]$ from right to left. Indeed the δ-subword graph of the reverse pattern recognizes at least all the δ-suffixes of the reverse pattern from right to left and thus at least all the δ-prefixes of the pattern from left to right.

```
Algorithm δ-BM2;
    i ← m;
    while i ≤ n
        q ← v0; j ← i; b ← 0;
        while (q, text[j], p) ∈ E
            q ← p; j ← j − 1;
            if q ∈ F then b ← i − j;
        if i − j > m then check and report
            δ-occurrence at position i − m + 1;
            i ← i + δ-per(pat);
        else i ← i + m − b;
```

The value δ-$per(pat)$ 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.

```
Algorithm δ-BM3;
    i ← m;
    while i ≤ n
        if hash(text[i − k + 1...i]) ∈ M_{δ,γ}(pat, k)
            then NAIVE(i, i + m − k − 1);
        i ← i + m − k;
```

4 Average time analysis of algorithms δ-BM1 and δ-BM3

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

Lemma 2. *The overall average time complexity of δ-BM1 and δ-BM3 algorithms is at most*

$$\frac{n}{m-k}\left(\left(q_{k,pat}+(1-q_{k,pat})\frac{p}{1-p}\right)k+(1-q_{k,pat})\,m+(1-q_{k,pat})\frac{1}{(1-p)^2}\right)$$

Proof. Each iteration of the algorithms tries to find occurrences of the pattern ending at positions $i..i+m-k-1$. We call those positions a window. The window for next iteration starts just behind the window for the previous iteration. The probability that the pattern is moved to the next window after at most k comparisons is exactly $q_{k,pat} = RR_k(\mathcal{D}_{pat})$. Now it is enough to prove that the average number of comparisons made by the naive algorithm in one iteration is bounded by

$$m+\frac{p}{1-p}k+\frac{1}{(1-p)^2}.$$

Let p_j, for $j = 1..m-k-1$ be the average number of comparisons made by the naive algorithm at position $i+j-1$ of the window. Since the we cannot assume that text symbols at positions $i-k+1..i$ are random (because $\mathcal{D}_{pat}(text[i-k+1..i]) = true$), we assume that the probability of matching those symbols with the symbols of the pattern during the naive algorithm is 1. Hence

$$p_1 \le (1-p)(k+1)+p(1-p)(k+2)+p^2(1-p)(k+3)+\cdots+$$
$$+p^{m-k-1}(1-p)(m-1)+p^{m-k}m.$$

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

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

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

$$p_i \le (1-p)+p(1-p)+\cdots+p^{i-2}(1-p)+$$
$$+p^{i-1}(1-p)(k+1)+p^i(1-p)(k+2)+\ldots p^{m-k-1}(1-p)(m-1)+p^{m-k}m$$

It can be proved that $p_1 \le k+\frac{1}{1-p}$ and $p_2 \le (1-p)+p(k+\frac{1}{1-p})$ and generally $p_i \le (1-p)+p(1-p)+\cdots+p^{i-2}(1-p)+p^{i-1}(k+\frac{1}{1-p})$. Hence, after some calculations we get

$$\sum p_i \le m+\frac{p}{1-p}k+\frac{1}{(1-p)^2}.$$

This completes the proof. □

As a corollary of previous lemma we have.

Theorem 2. *Let $k \le 0.99m$ and $p \le 0.99$. The average time complexity of the algorithms δ-BM1 and δ-BM3 is*

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

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, $1 - q_{k,pat}$ is the probability that a random string of length k is not a subword of pat. The number of subwords of length k of pat is at most m and the number of all words of length k is m^2 so $1 - q_{k,pat} \leq \frac{1}{m}$ thus the average time complexity is $O(\frac{n}{m} \log m)$ the best possible. Moreover k may depend also on the pattern pat itself. If pat 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 $1 - q_{k,pat}$ decreases below an acceptable level.

5 Experimental Results

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 [5] 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 were 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 4 to 5. 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 and better than δ-Tuned-Boyer-Moore algorithm (which is the best among the known algorithms) for large values of m ($m \geq 20$). For larger 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 set of parameters. A typical symbol could be $(a_1, a_2, a_3, ...a_k)$, where a_1 represents the pitch, a_3 represents the duration, a_4 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.

References

1. E. Cambouropoulos, T. Crawford and C.S. Iliopoulos, Pattern Processing in Melodic Sequences: Challenges, Caveats and Prospects, In G. Wiggins, editor, *Proceedings of the Artificial Intelligence and Simulation of Behaviour Symposium*, The Society for the Study of Artificial Intelligence and Simulation of Behaviour, Edinburgh, UK, pp 42-47, 1999.
2. E. Cambouropoulos, M. Crochemore, C. S. Iliopoulos, L. Mouchard and Y. J. Pinzon, Algorithms for computing approximate repetitions in musical sequences. In R. Raman and J. Simpson, editors, *Proceedings of the 10th Australasian Workshop On Combinatorial Algorithms*, Perth, WA, Australia, pp 129–144, 1999.
3. T. Crawford, C. S. Iliopoulos and R. Raman, String Matching Techniques for Musical Similarity and Melodic Recognition, *Computing in Musicology* **11** (1998) 73–100.
4. M. Crochemore, A. Czumaj, L. Gąsieniec, S. Jarominek, T. Lecroq, W. Plandowski and W. Rytter, Speeding-up two string matching algorithms, *Algorithmica* **12** (4/5) (1994) 247–267.
5. M. Crochemore, C.S. Iliopoulos, T. Lecroq and Y.J. Pinzon, Approximate string matching in musical sequences, In M. Balík and M. Simánek, editors, *Proceedings of the Prague Stringology Conference'01*, Prague, Tcheque Republic, 2001, Annual Report DC–2001–06, 26–36.
6. V. Fischetti, G. Landau, J.Schmidt and P. Sellers, Identifying periodic occurrences of a template with applications to protein structure, *Proceedings of the 3rd Combinatorial Pattern Matching*, Lecture Notes in Computer Science 644, pp. 111–120, 1992.
7. S. Karlin, M. Morris, G. Ghandour and M.-Y. Leung, Efficient algorithms for molecular sequences analysis, *Proc. Natl. Acad. Sci. USA* **85** (1988) 841–845.
8. P. McGettrick, MIDIMatch: Musical Pattern Matching in Real Time, MSc Dissertation, York University, UK, 1997.
9. A. Milosavljevic and J. Jurka, Discovering simple DNA sequences by the algorithmic significance method, *Comput. Appl. Biosci.* **9** (1993) 407–411.
10. P. A. Pevzner and W. Feldman, Gray Code Masks for DNA Sequencing by Hybridization, *Genomics* **23** (1993) 233-235.
11. P. Y. Rolland and J. G. Ganascia, Musical Pattern Extraction and Similarity Assessment, In E. Miranda, editor, *Readings in Music and Artificial Intelligence*, Harwood Academic Publishers, 1999.

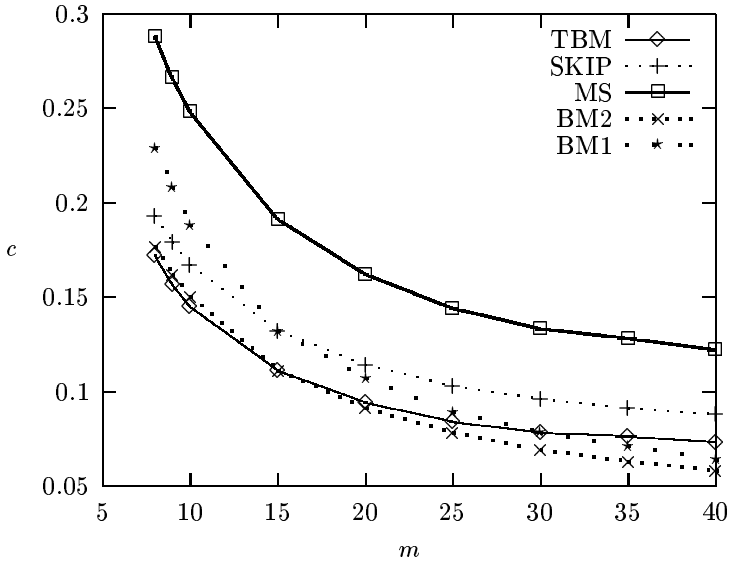

Fig. 4. Results for $\delta = 1$.

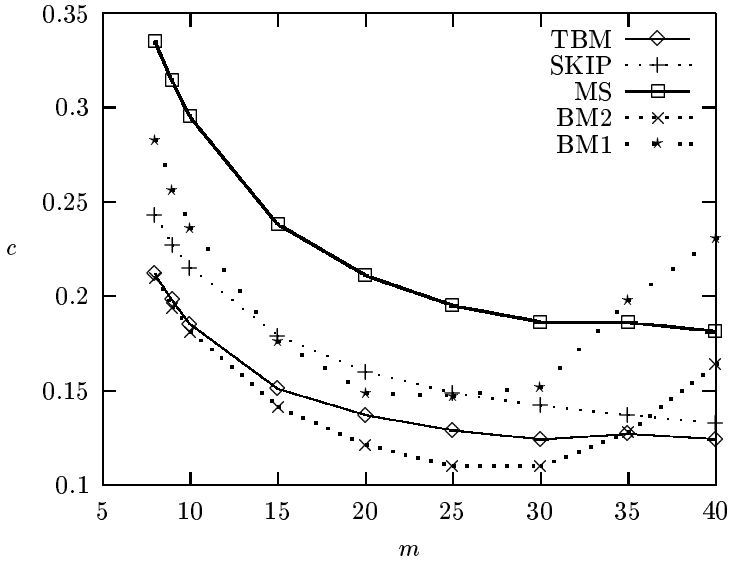

Fig. 5. Results for $\delta = 2$.