A Bit-parallel Suffix Automaton Approach for (δ, γ)-Matching in Music Retrieval

Maxime Crochemore[1,2⋆], Costas S. Iliopoulos[2], Gonzalo Navarro[3⋆⋆], and Yoan J. Pinzon[2,4⋆⋆]

[1] Institut Gaspard-Monge, Université de Marne-la-Vallée, France
mac@univ-mlv.fr
www-igm.univ-mlv.fr/∼mac

[2] Dept. of Computer Science, King's College, London, England
{csi,pinzon}@dcs.kcl.ac.uk
www.dcs.kcl.ac.uk/staff/csi, www.dcs.kcl.ac.uk/staff/pinzon

[3] Dept. of Computer Science, University of Chile, Chile
gnavarro@dcc.uchile.cl
www.dcc.uchile.cl/∼gnavarro

[4] Laboratorio de Cómputo Especializado, Universidad Autónoma de Bucaramanga, Colombia

Abstract. (δ, γ)-Matching is a string matching problem with applications to music retrieval. The goal is, given a pattern $P_{1\ldots m}$ and a text $T_{1\ldots n}$ on an alphabet of integers, find the occurrences P' of the pattern in the text such that (i) $\forall 1 \leq i \leq m$, $|P_i - P'_i| \leq \delta$, and (ii) $\sum_{1 \leq i \leq m} |P_i - P'_i| \leq \gamma$. Several techniques for (δ, γ)-matching have been proposed. In this paper we show that a classical string matching technique that combines bit-parallelism and suffix automata can be successfully adapted to this problem. This is the first character-skipping algorithm that skips characters using both δ and γ. We implemented our algorithm and drew experimental results on real music showing that our algorithm is superior to current alternatives.

1 Introduction

The string matching problem is to find all the occurrences of a given pattern $P_{1\ldots m}$ in a large text $T_{1\ldots n}$, both being sequences of characters drawn from a finite character set Σ. This problem is fundamental in computer science and is a basic need of many applications, such as text retrieval, music retrieval, computational biology, data mining, network security, etc. Several of these applications require, however, more sophisticated forms of searching, in the sense of extending the basic paradigm of the pattern being a simple sequence of characters.

In this paper we are interested in music 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). It is known that exact matching cannot be

⋆ Partly supported by CNRS and NATO.
⋆⋆ Supported by CYTED VII.19 RIBIDI Project.

used to find occurrences of a particular melody, so one resorts to different forms of *approximate* matching, where a limited amount of *differences* of diverse kinds are permitted between the search pattern and its occurrence in the text.

The approximate matching problem has been used for a variety of musical applications [15, 9, 19, 20, 6]. 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 permit a difference of at most δ units between the pattern character and its corresponding text character in an occurrence, 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. Additionally, we require that the total number of differences across all the pattern positions does not exceed γ, in order to limit the total number of differences while keeping sufficient flexibility at individual positions.

The formalization of the above problem is called (δ, γ)-matching. The problem is defined as follows: the alphabet Σ is assumed to be a set of integer numbers, $\Sigma \subset \mathbb{Z}$. Apart from the pattern P and the text T, two extra parameters, $\delta, \gamma \in \mathbb{N}$, are given. The goal is to find all the occurrences P' of P in T such that (i) $\forall 1 \leq i \leq m,\ |P_i - P'_i| \leq \delta$, and (ii) $\sum_{1 \leq i \leq m} |P_i - P'_i| \leq \gamma$.

Several recent algorithms exist to solve this problem [7, 10, 8, 11]. Some are based on extending well-known paradigms such as the Boyer-Moore family or the use of suffix automata. Others are based on bit-parallelism. We detail them in the next section. On the other hand, it was shown in [17, 18] that bit-parallelism and suffix automata can be nicely combined in order to obtain faster, simpler, and more flexible algorithms, which are especially robust to handle extended string matching problems (classes of characters, wild cards, regular expressions, approximate searching based on Hamming or edit distance, and so on).

In this paper we extend the bit-parallel suffix automata to handle (δ, γ)-matching: The resulting algorithm is extremely simple and much faster than the existing approaches. It is also the first truly (δ, γ) character-skipping algorithm, as it skips characters using both criteria. Existing approaches do just δ-matching and check the candidates for the γ-condition.

We use the following definitions throughout the paper. A word $x \in \Sigma^*$ is a *factor* (or substring) of P if P can be written $P = uxv$, $u, v \in \Sigma^*$. A factor x of P is called a *suffix* (*prefix*) of P if $P = ux$ $(P = xu)$, $u \in \Sigma^*$. The number of bits in the computer word is denoted w.

2 Related Work

2.1 (δ, γ)-Matching

We recall three approaches that have been attempted to (δ, γ)-matching.

Bit-Parallelism consists of taking advantage of the intrinsic parallelism of the bit operations inside a computer word [1], so as to pack several values in a single word and manage to update them all in less operations than those necessary to update the values separately. In [7, 8] this approach was used to obtain an $O(n)$ search time algorithm for (δ, γ)-matching called Shift-Plus. The algorithm packs m counters whose maximum value is $m\delta$, so it can pack all them in a single computer word provided $m\lceil\log_2(1 + m\delta)\rceil \leq w$. Otherwise, several computer words have to be maintained, for a total search time of $O(n\ m \log(m\delta)/w)$.

Occurrence Heuristics consist of skipping some text characters by using information on the position of some characters in the pattern. Typical algorithms of this type are those of the Boyer-Moore family [5, 21]. In [7], several algorithms of this type were proposed for δ-matching (a restricted case where $\gamma = \infty$), and they were extended to general (δ, γ)-matching in [10]. These are Tuned-Boyer-Moore, Skip-Search and Maximal-Shift, each of which have a counterpart in exact string matching. It is shown that these algorithms are faster than the bit-parallel ones, as they are simple and able to skip text characters.

Substring Heuristics consist of skipping some text characters by using information on the position of some pattern substrings. Typical algorithms of this type are those based on suffix automata [13, 12]. In [10, 11] three algorithms based on these ideas, called δ-BM1, δ-BM2 and δ-BM3, are proposed. They try to generalize the suffix automata to δ-matching, but they obtain only an approximation that accepts more occurrences than necessary, which have to be verified later. In classical string matching, substring heuristics perform better than character heuristics on small alphabets. This makes it probable that in this application substrings heuristics perform better for large δ and γ values.

2.2 Bit-parallel Suffix Automata

Bit-parallelism provides a general method to use automata in their nondeterministic form rather than converting them to deterministic. The latter is the classical approach and normally involves a complex construction algorithm and lack of flexibility in the resulting scheme (see the previous comment on adapting suffix automata to δ-matching). Nondeterministic automata, on the other hand, tend to be rather simple and can be easily extended to handle new problems. Bit-parallelism permits simulating nondeterministic automata as they are, since they can handle all the active states in a single operation.

In this spirit, the algorithm BNDM was developed in [17] as a combination between Shift-Or [2] (a bit-parallel algorithm) and BDM [13] (an algorithm based on suffix automata and able to skip characters). The result is an algorithm with the best of both worlds: simple, efficient, and extensible. It is shown that it outperforms both Shift-Or and BDM, and that there is no reason for bit-parallel algorithms not to skip characters. BNDM was extended to handle classes of characters, wild cards, regular expressions, and widely used forms of approximate searching [18].

2.3 Our Work in Context

Our goal in this paper is to develop an extension of BNDM to handle (δ, γ)-matching. The algorithm turns out to be simple and very efficient. In the above categorization, it corresponds to a crossing between bit-parallel and substring-heuristic algorithms. Compared to the original bit-parallel algorithm [7,8], it makes a better packing of values, since it needs only $m\lceil 1 + \log_2(\gamma + 1)\rceil$ bits, so the number of characters inspected has to be multiplied by $O(m \log(\gamma)/w)$. Compared to the original substring matching heuristics, the nondeterministic version is able to accept exactly the patterns that qualify, without any need of further verification. In particular, all the existing methods really do δ-matching and enforce the γ-condition in a further verification, while we are able of enforcing both conditions as we scan the text. This makes up a much more robust and efficient algorithm.

3 Searching with Suffix Automata

We describe in this section the BDM pattern matching algorithm [12,13], which is based on a suffix automaton. A *suffix automaton* on a pattern $P_{1\dots m}$ (frequently called DAWG(P), for Deterministic Acyclic Word Graph) is the minimal (incomplete) deterministic finite automaton that recognizes all the suffixes of this pattern. By "incomplete" we mean that unnecessary transitions are not present. The nondeterministic version of this automaton has a very regular structure and is shown in Figure 1.

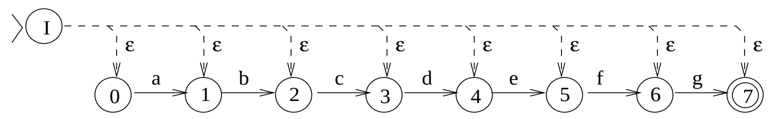


Fig. 1. A nondeterministic suffix automaton for the pattern P = `"abcdefg"`. Dashed lines represent ε-transitions (i.e. they occur without consuming any input). I is the initial state of the automaton

The (deterministic) suffix automaton is a well known structure [12]. The size of DAWG(P) is linear in m (counting both nodes and edges), and a linear online construction algorithm exists [12]. A very important fact for our algorithm is that this automaton cannot only be used to recognize the suffixes of P, but also factors of P: The automaton has active states as long as we have read a factor of P.

The suffix automaton structure is used in [12, 13] to design a simple pattern matching algorithm called BDM. This algorithm is $O(mn)$ time in the worst case, but optimal on average ($O(n \log_{|\Sigma|} m/m)$ time). To search for P in a text T, the suffix automaton of $P^r = P_m P_{m-1} \ldots P_1$ (i.e., the pattern read backwards) is built. A window of length m is slid along the text, from left to right. The algorithm searches the window backwards for a factor of the pattern P using the suffix automaton. During this search, if a terminal state is reached which does not correspond to the entire pattern P, the window position is recorded (in a variable *last*). This corresponds to finding a *prefix* of the pattern starting at position *last* inside the window and ending at the end of the window (since the suffixes of P^r are the reverse prefixes of P). Since we remember the last prefix recognized backwards, we have the *longest* prefix of P that is a suffix of the window. This backward search ends in two possible forms:

1. We fail to recognize a factor, i.e., we reach a character σ that does not correspond to a transition in DAWG(P^r). Figure 2 illustrates this case. In this case we shift the window to the right, its starting position corresponding to the position *last* (we cannot miss an occurrence because in that case the suffix automaton would have found its prefix in the window).

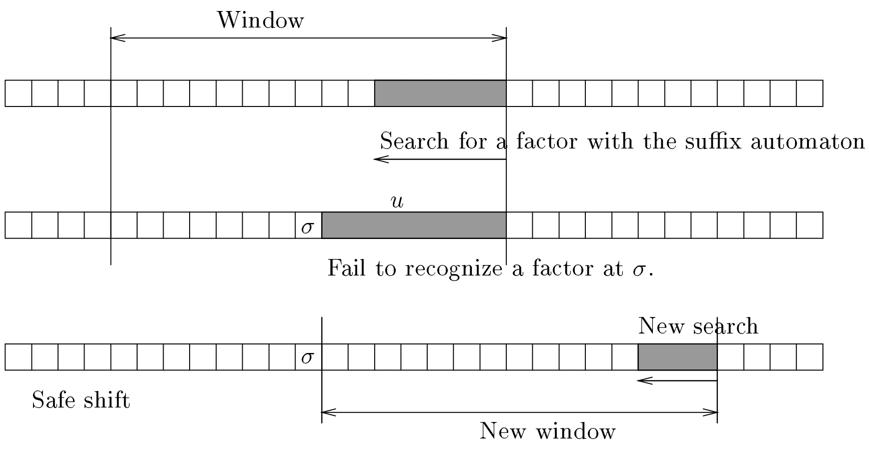

Window
Search for a factor with the suffix automaton
u
σ
Fail to recognize a factor at σ.
New search
σ
Safe shift
New window

Fig. 2. Basic search with the suffix automaton

2. We reach the beginning of the window, therefore recognizing the pattern P. We report the occurrence, and shift the window exactly as in the previous case (notice that we have the previous *last* value).

4 Our Algorithm

We first describe a forward-scan version that extends Shift-And and permits us explaining the details of the bit-parallel simulation, and then a backward-scanning version that extends BNDM.

We start with some terminology. A *bit mask* of length r is a sequence of bits, denoted $b_r \dots b_1$. We use exponentiation to denote bit repetition (e.g. $0^3 1 = 0001$). We use C-like syntax for operations on the bits of computer words: "|" is the bitwise-or, "&" is the bitwise-and, "^" is the bitwise-xor and "~" complements all the bits. The shift-left operation, "<<", moves the bits to the left and enters zeros from the right. The shift-right, ">>" moves the bits in the other direction. Finally, we can perform arithmetic operations on the bits, such as addition and subtraction, which operate the bits as if they formed a number. For instance, $b_r \dots b_x 10000 - 1 = b_r \dots b_x 01111$.

4.1 Forward Scanning

The Shift-And algorithm first builds a table B which for each character stores a bit mask $b_m \dots b_1$. The mask in $B[c]$ has the i-th bit set if and only if $P_i = c$. The state of the search is kept in a machine word $D = d_m \dots d_1$, where d_i is set whenever $P_{1 \dots i}$ matches the end of the text read up to now. Therefore, we report a match whenever d_m is set.

We set $D = 0^m$ originally and, for each new text character T_j, update D using the formula

$$D \;\leftarrow\; ((D << 1) \mid 0^{m-1}1) \;\;\&\;\; B[T_j]$$

We now extend the Shift-And algorithm. First of all, notice that δ-matching is trivial under the bit-parallel approach, as it can be accommodated using the ability to search for classes of characters. We define that pattern character c matches text characters $c - \delta \dots c + \delta$, therefore setting the i-th bit of $B[c]$ to 1 if and only if $|P_i - c| \leq \delta$. The rest of the algorithm is unchanged. On a uniform distribution over $\Sigma = \{1 \dots |\Sigma|\}$ we obtain $O(n \log_{|\Sigma|/\delta}(m)/m)$ time for the BNDM version, and we still need $\lceil m/w \rceil$ computer words for the simulation. However, the real challenge is to do (δ, γ)-matching. In the following we assume $\delta \leq \gamma \leq m\delta$, otherwise the formulation makes little sense.

Instead of storing just one bit d_i to tell whether $P_{1 \dots i}$ matches $T_{j-i+1 \dots j}$, we store a counter c_i to record the sum of the absolute differences between the corresponding characters. That is

$$c_i \;=\; \sum_{1 \leq k \leq i} |P_k - T_{j-i+k}|$$

In fact we are only interested in storing $\min(c_i, \gamma + 1)$, as any value larger than γ is equivalent for us. For reasons that will be clear soon, we need to represent c_i such that its highest bit is set to 1 if and only if $c_i > \gamma$. So we use $\ell = 1 + \lceil \log_2(\gamma + 1) \rceil$ bits to represent c_i, and instead of representing c_i

we represent $c_i + 2^{\ell-1} - (\gamma + 1)$. This guarantees that the highest bit is set when c_i reaches $\gamma+1$ (as its representation reaches $2^{\ell-1}$). Hence our bit mask D needs $m\ell = m(1 + \lceil \log_2(\gamma+1) \rceil)$ bits and our simulation needs $O(m \log(\gamma)/w)$ computer words.

We precompute a mask $B[c]$ of counters as follows. The i-th counter of $B[c]$ will store $|P_i - c|$ if and only if $|P_i - c| \leq \delta$. Otherwise, the characters simply do not match, in which case we store $\gamma + 1$ for this counter. This value ensures that no match will be reported, as the global count of differences will surpass γ. Since $\delta \leq \gamma$, ℓ bits suffice to store each of these counters.

The algorithm is basically the same of Shift-And, except that we add $B[c]$ to D in order to keep count of the sum of the differences between the matched characters. The overflow is avoided as follows: we remove the highest bits from the counters in D before adding those of $B[T_j]$, and then restore them in the result. Therefore, (1) overflow is impossible because we are adding two values that at most add up $2\gamma+1$, and we have enough space to store $2^\ell - 1 \geq 2(\gamma+1) - 1 = 2\gamma + 1$ differences; (2) if the highest bit was set before, it will stay set; (3) if the highest bit was not yet set then our operation with highest bits does not affect the sum. Note that it is not strictly true that we maintain $\min(c_i, \gamma+1)$, but it is true that the highest bit of each counter i is set if and only if $c_i > \gamma$, and this is enough for the correctness of the algorithm.

This solution has some resemblances with that of [3] for Hamming distance.

Figure 3 depicts the forward-scanning algorithm. It is $O(n)$ time if $m \log \gamma = O(w)$, otherwise it takes time $O(nm \log(\gamma)/w)$. The preprocessing takes $O(m|\Sigma|)$ time. We remark that previous forward scanning versions [7, 8] required $O(nm \log(m\delta)/w)$ bits, which is strictly larger than our requirement. The difference is that we managed to keep the counters below 2γ instead of letting them grow up to $m\delta$.

4.2 Backward Scanning

We start by explaining the BNDM algorithm [17] and then show how to extend it. We assume $m \leq w$ in the exposition for simplicity, although the scheme is general.

The BNDM algorithm moves a window over the text. Each time the window is positioned at a new text position just after *pos*, it searches backwards the window $T_{pos+1\ldots pos+m}$ using the DAWG automaton, until either m iterations are performed (which implies a match in the current window) or the automaton cannot follow any transition. In this case, the bit d_i at iteration k is set if and only if $P_{m-i+1\ldots m-i+k} = T_{pos+1+m-k\ldots pos+m}$. Some observations follow

- Since we begin at iteration 0, the initial value for D is 1^m (recall that we use exponentiation to denote bit repetition).
- There is a match if and only if after iteration m it holds $d_m = 1$.
- Whenever $d_m = 1$, we have matched a prefix of the pattern in the current window. The longest prefix matched (excluding the complete pattern) corresponds to the next window position (variable *last*).

```
Forward-Scan ($P_{1 \dots m}$, $T_{1 \dots n}$, $\delta$, $\gamma$)
    Preprocessing
        $\ell \leftarrow 1 + \lceil \log_2(\gamma+1) \rceil$
        For $c \in \Sigma$ Do
            $B[c] \leftarrow 0^m$
            For $i \in 1 \dots m$ Do
                $B[c] \leftarrow B[c] \mid ((|c - P_i| > \delta ? \gamma + 1 : |c - P_i|) << (\ell(i-1)))$
    Search
        $D \leftarrow 1^{m\ell}$
        For $j \in 1 \dots n$ Do
            If $D$ & $10^{m\ell-1} = 0^{m\ell}$ Then
                Report an occurrence at $j - m + 1$
            $D \leftarrow (D << \ell) \mid (2^{\ell-1} - (\gamma+1))$
            $H \leftarrow D$ & $(10^{\ell-1})^m$
            $D \leftarrow ((D$ & $\sim H) + B[T_j]) \mid H$
```

Fig. 3. Forward scanning algorithm for (δ, γ)-matching

– Since there is no initial self-loop, this automaton eventually runs out of active states. Moreover, states $(m-k) \dots m$ are inactive at iteration k.

The algorithm works as follows. Every time we position the window in the text we initialize D and scan the window backwards. For each new text character we update D. Each time we find a prefix of the pattern $(d_m = 1)$ we remember the position in the window. If we run out of 1's in D then there cannot be a match and we suspend the scanning (this corresponds to not having any transition to follow in the automaton). If we can perform m iterations then we report a match.

We use a mask B which stores a bit mask for each character c. This mask sets the bits corresponding to the positions i where $P_i = c$ (just as in Shift-And). The formula to update D is

$$D \quad \leftarrow \quad (D \ \& \ B[T_j]) \quad << \quad 1$$

We now extend BNDM to (δ, γ)-matching. The main differences with respect to the representation used in forward scanning are (1) we initialize the counters of D to $c_i = 0$ because they correspond to matching empty strings; (2) after shifting D, the fresh counters that enter from the right are not important (the important ones are those present when we start scanning the window); and (3) we suspend scanning the window when all the counters exceeded γ.

Figure 4 depicts the backward-scanning algorithm. This is the first character-skipping algorithm that does not use verifications and is able to stop scanning text windows that δ-match the pattern, if they do not (δ, γ)-match the pattern.

```
Backward-Scan ($P_{1\ldots m}$, $T_{1\ldots n}$, $\delta$, $\gamma$)
Preprocessing
    $\ell \leftarrow 1 + \lceil \log_2(\gamma+1) \rceil$
    For $c \in \Sigma$ Do
        $B[c] \leftarrow 0^m$
        For $i \in 1 \ldots m$ Do
            $B[c] \leftarrow B[c] \mid (|c - P_{m-i+1}| > \delta ? \gamma+1 : |c - P_{m-i+1}|) << (\ell(i-1)))$
Search
    $pos \leftarrow 0$
    While $pos \leq n - m$ Do
        $j \leftarrow m,\ last \leftarrow m$
        $D \leftarrow (2^{\ell-1} - (\gamma+1)) \times (0^{\ell-1}1)^m$
        While $D \ \& \ (10^{\ell-1})^m \neq (10^{\ell-1})^m$ Do
            $H \leftarrow D \ \& \ (10^{\ell-1})^m$
            $D \leftarrow ((D \ \& \ \sim H) + B[T_j]) \mid H$
            $j \leftarrow j - 1$
            If $D \ \& \ 10^{m\ell-1} = 0^{m\ell}$ Then
                If $j > 0$ Then $last \leftarrow j$
                Else Report an occurrence at $pos + 1$
            $D \leftarrow (D << \ell) \mid 10^{\ell-1}$
        $pos \leftarrow pos + last$
```

Fig. 4. Backward scanning algorithm for (δ,γ)-matching. Some code optimizations are not included for simplicity

5 Experimental Results

In this section we show experimental evidence supporting the superiority of the new algorithm (This) compared to the (δ,γ)-Boyer-Moore algorithm (BM2) presented in [10, 11], which is currently the most competitive choice.

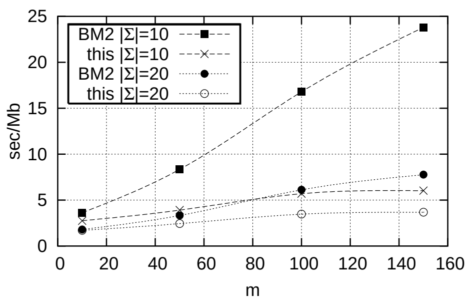
The time reported includes only the searching phase. Preprocessing was negligible. The tests were performed using a SUN Ultra Enterprise 300MHz running Solaris Unix with $w = 32$. We used the GNU `g++` compiler version 2.95.1. Each data point represents the median of 60 trials.

We ran our experiments using both real music and random text. The music data used for this study comes from a data base of MIDI files of classic music with 1.8Mb of absolute pitches. We also make use of this music data base to measure the zero-order and one-order entropy to estimated the size of alphabet needed to emulated music using random text. Zero-order entropy was equivalent to having a random alphabet of size 17.35. One-order entropy was much smaller, 6.27. Therefore, we used random text uniformly distributed with alphabet size of 10–20 for this study. Other typical parameter values were 0–5 for δ, $1.5m$–$2.0m$ for γ, and 10–200 for m.

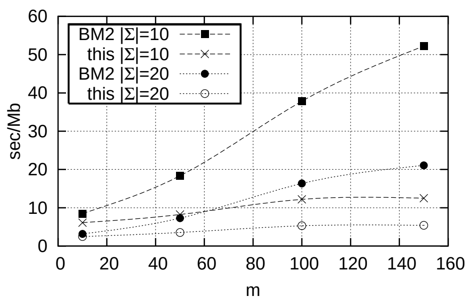
Figure 5 shows plots of the performance of both algorithms using random data. For the different combinations of δ and γ used in these experiments, our algorithm (This) was significantly faster than Algorithm BM2. As expected, the performance of the algorithm degrades with smaller alphabets. However, it also degrades as m increases, as the implementation is limited to using m/w number of computer words and skipping at most w characters. To speed-up the matching algorithm we can use an alphabet reduction method such as octave equivalence [14].

The results using real music data are shown in Figure 6. Although the difference is smaller than on synthetic data, clearly This algorithm performs better. As can be seen, the dependence on δ is significant to the extent that it can double (note the change of scale) the time it takes by going from δ =2 to δ =4. The dependence on γ, on the other hand, is not much significant.

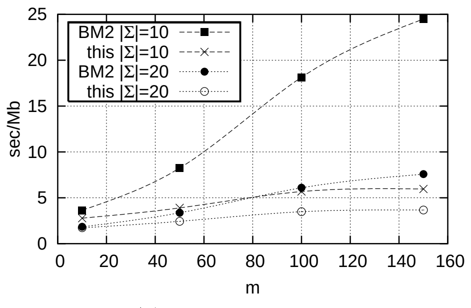
In conclusion the algorithm introduced in this paper performs consistently better than previous known algorithms.



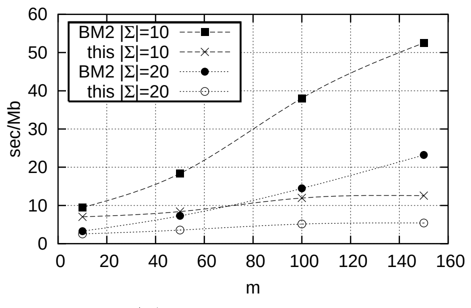
(a) $\delta = 2$ and γ=1.5m



(b) $\delta = 4$ and γ=1.5m

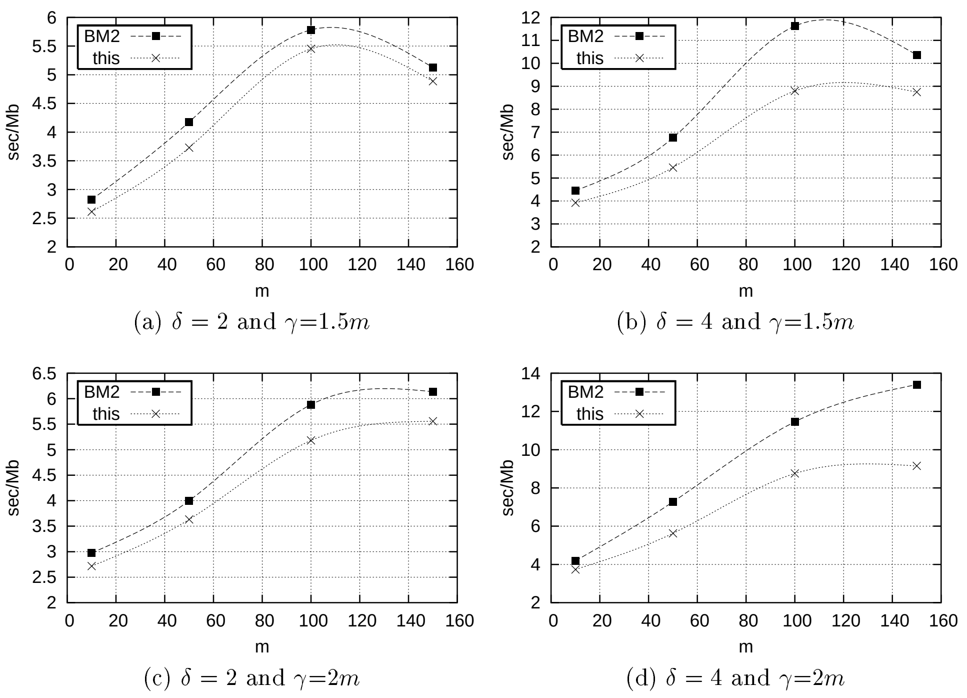


(c) $\delta = 2$ and γ=2m



(d) $\delta = 4$ and γ=2m

Fig. 5. Timing figures for random data



(a) $\delta = 2$ and γ=$1.5m$

(b) $\delta = 4$ and γ=$1.5m$

(c) $\delta = 2$ and γ=$2m$

(d) $\delta = 4$ and γ=$2m$

Fig. 6. Timing figures for real music data

6 Conclusions

We have presented a new bit-parallel algorithm for (δ, γ)-matching, an extended string matching problem with applications in music retrieval. Our new algorithm is a crossing between bit-parallelism and suffix automata and has several advantages over the previous approaches: it makes better use of the bits of the computer word, it inspects less text characters, it is simple and extensible.

Our algorithms is also the first truly (δ, γ) character-skipping algorithm, as it skips characters using both criteria. Existing approaches do just δ-matching and check the candidates for the γ-condition. This makes our algorithm a stronger choice for this problem.

The algorithm we have presented is useful for short pattern lengths, as it is limited by the length of the computer word. We have handled longer patterns with the naive approach of using as many computer words as needed to represent all the counters. A more sophisticated approach we are pursuing is to partition the pattern into pieces short enough to be handled with the basic algorithm. It is interesting to notice that if we partition the pattern into j pieces, then at least one of them has to match with $\gamma' = \lfloor \gamma/j \rfloor$ differences overall, so we do (δ, γ')-matching in the pieces. Moreover, if $\delta > \gamma'$ we do (γ', γ')-matching. Hence we run j searches for shorter patterns and check every match of a piece for a complete occurrence. These pieces can be grouped and searched for together using the so-called "superimposition". These ideas have been used in [4, 16] for approximate string matching, and should be useful here too.

It is not hard to design an algorithm with the same average complexity but also linear in the worst case, as done in [17]. Despite theoretically interesting, this improvement is usually disregarded because it worsens the practical performance of the algorithm.

A more challenging problem is to consider text indexing approaches, that is, preprocessing the musical strings in order to permit fast searching of patterns later. A simple solution is the use of a suffix tree of the text combined with backtracking, which yields search times which are exponential on the pattern length but independent of the text length [22].

We also plan to investigate further on more sophisticated matching problems that arise in music retrieval. For example, it would be good to extend (δ, γ)-matching in order to permit insertions and deletions of symbols.

References

1. R. Baeza-Yates. Text retrieval: Theory and practice. In *12th IFIP World Computer Congress*, volume I, pages 465–476. Elsevier Science, September 1992.
2. R. Baeza-Yates and G. Gonnet. A new approach to text searching. *Comm. ACM*, 35(10):74–82, October 1992.
3. R. Baeza-Yates and G. Gonnet. Fast string matching with mismatches. *Information and Computation*, 108(2):187–199, 1994.
4. R. Baeza-Yates and G. Navarro. Faster approximate string matching. *Algorithmica*, 23(2):127–158, 1999.
5. R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.
6. E. Cambouropoulos, T. Crawford, and C. Iliopoulos. Pattern processing in melodic sequences: Challenges, caveats and prospects. In *Proc. Artificial Intelligence and Simulation of Behaviour (AISB'99) Convention*, pages 42–47, 1999.
7. E. Cambouropoulos, M. Crochemore, C. Iliopoulos, L. Mouchard, and Y. J. Pinzon. Algorithms for computing approximate repetitions in musical sequences. In *Proc. 10th Australasian Workshop on Combinatorial Algorithms (AWOCA'99)*, pages 129–144, 1999.
8. E. Cambouropoulos, M. Crochemore, C. S. Iliopoulos, L. Mouchard, and Y. J. Pinzon. Algorithms for computing approximate repetitions in musical sequences. *Int. J. Comput. Math.*, 79(11):1135–1148, 2002.
9. T. Crawford, C. Iliopoulos, and R. Raman. String matching techniques for musical similarity and melodic recognition. *Computing in Musicology*, 11:73–100, 1998.
10. M. Crochemore, C. Iliopoulos, T. Lecroq, Y. J. Pinzon, W. Plandowski, and W. Rytter. Occurence and substring heuristics for δ-matching. *Fundamenta Informaticae*, 55:1–15, 2003.
11. M. Crochemore, C. Iliopoulos, T. Lecroq, W. Plandowski, and W. Rytter. Three heuristics for δ-matching: δ-bm algorithms. In Combinatorial Pattern Matching, CPM'2002, LNCS v. 2373, pages 178–189. Springer-Verlag, 2002.
12. M. Crochemore and W. Rytter. *Text algorithms.* Oxford University Press, 1994.

13. A. Czumaj, M. Crochemore, L. Gasieniec, S. Jarominek, Thierry Lecroq, W. Plandowski, and W. Rytter. Speeding up two string-matching algorithms. *Algorithmica*, 12:247–267, 1994.
14. K Lemström and J. Tarhio. Searching monophonic patterns within polyphonic sources. In *Proc. of Content-Based Multimedia Information Access*, volume 2, pages 1261–1279, 2000.
15. P. McGettrick. *MIDIMatch: Musical Pattern Matching in Real Time.* MSc. Dissertation, York University, U.K., 1997.
16. G. Navarro and R. Baeza-Yates. Improving an algorithm for approximate string matching. *Algorithmica*, 30(4):473–502, 2001.
17. G. Navarro and M. Raffinot. Fast and flexible string matching by combining bit-parallelism and suffix automata. *ACM Journal of Experimental Algorithmics (JEA)*, 5(4), 2000.
18. G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings – Practical on-line rch algorithms for texts and biological sequences.* Cambridge University Press, 2002. ISBN 0-521-81307-7.
19. P. Roland and J. Ganascia. Musical pattern extraction and similarity assessment. In E. Miranda, editor, *Readings in Music and Artificial Intelligence*, pages 115–144. Harwood Academic Publishers, 2000.
20. L. A. Smith, E. F. Chiu, and B. L. Scott. A speech interface for building musical score collections. In *Proc. of the fifth ACM conference on Digital libraries*, pages 165–173. ACM Press, 2000.
21. D. Sunday. A very fast substring searching algorithm. *Comm. ACM*, 33(8):132–142, August 1990.
22. E. Ukkonen. Approximate string matching over suffix trees. In *Proc. 4th Annual Symposium on Combinatorial Pattern Matching (CPM'93)*, pages 228–242, 1993.