Nordic Journal of Computing

APPROXIMATE STRING MATCHING WITH GAPS

MAXIME CROCHEMORE
Institut Gaspard Monge, Université de Marne-la-Vallée,
CEDEX 2, France.
e-mail: mac@univ-mlv.fr

COSTAS ILIOPOULOS
Dept. Computer Science, King's College London
London WC2R 2LS, England, and
School of Computing, Curtin University of Technology, GPO Box 1987 U, WA.
e-mail:csi@dcs.kcl.ac.uk

CHRISTOS MAKRIS
Computer Engineering & Informatics Dept. of University of Patras and
Computer Technology Institute (CTI),
Rio, Greece, P.O. BOX 26500.
e-mail: makri@ceid.upatras.gr

WOJCIECH RYTTER
Uniwersytet Warszawski, Banacha2, 02-097, Warszawa, Poland and
Department of Computer Science, University of Liverpool,
Liverpool L69 7ZF, UK.
e-mail: rytter@csc.liv.ac.uk

ATHANASIOS TSAKALIDIS
Computer Engineering & Informatics Dept. of University of Patras and
Computer Technology Institute (CTI),
Rio, Greece, P.O. BOX 26500.
e-mail: tsak@cti.gr

KOSTAS TSICHLAS
Computer Engineering & Informatics Dept. of University of Patras and
Computer Technology Institute (CTI),
Rio, Greece, P.O. BOX 26500.
e-mail: tsihlas@ceid.upatras.gr

Abstract. In this paper we consider several new versions of approximate string matching with gaps. The main characteristic of these new versions is the existence of gaps in the matching of a given pattern in a text. Algorithms are devised for each version and their time and space complexities are stated. These specific versions of approximate string matching have various applications in computerized music analysis.

CR Classification: F.2.2

Key words: string algorithms, approximate string matching, dynamic programming, computerized music analysis

Received January 19, 2004.

1. Introduction

In the problem of pattern matching in strings one is interested in finding all occurrences of a pattern in a text. When we consider the approximate version of this problem we do not require a perfect matching but a matching that is good enough to satisfy certain criteria. The problem of finding substrings of a text similar to a given pattern has been extensively studied in recent years because it has a variety of applications including file comparison, spelling correction, information retrieval, searching for similarities among biosequences and computerized music analysis. One of the most common variants of the approximate string matching problem is that of finding substrings that match the given pattern with at most *k-differences*. In this case, k defines the approximation extent of the matching (the edit distance with respect to the three edit operations - *mismatch*, *insert*, *delete*). This paper focuses in one special type of approximate matching that mainly arises in musical information retrieval, i.e. *δ-approximate matching*. It is well known that a musical score can be represented as a string. This can be accomplished by defining the alphabet to be the set of notes in the chromatic or diatonic notation or the set of intervals that appear between notes. These algorithms can be easily used in the analysis of musical works in order to discover similarities between different musical entities that may lead to establishing a *characteristic signature* [4].

In addition, efficient algorithms for computing approximate matching and repetitions of substrings are also used in molecular biology [8, 13, 15] and particularly in DNA sequencing by hybridization, reconstruction of DNA sequences from known DNA fragments, in human organ and bone marrow transplantation as well as the determination of evolutionary trees among distinct species.

Because exact matching cannot help us to find occurrences of a particular melody in a musical work due to the transformation of the particular melody throughout the whole musical work we are compelled to use approximate matching that can absorb, to some extent, this transformation and report the occurrences of this melody. The transformation in different occurrences of a particular melody throughout a musical play is translated into errors of different occurrences of a substring with respect to an initial pattern. Quantity δ defines the error margins of such an approximation.

In [2], algorithms *Shift-And* and *Shift-Plus* were presented as efficient solutions to find all δ-occurrences of a given pattern in a text. The *Shift-And* algorithm is based on the constant time computation of different states for each symbol in the text by using bitwise techniques. The time complexities of the *Shift-And* and *Shift-Plus* algorithms are linear for m such that $m < \omega$ and $m \log(m \times \delta) < \omega$ respectively, where ω is the word size of the machine. In general, we can generalize them to work in $O(nm/\omega)$ and $O(nm \log(m \times \delta)/\omega))$ time respectively. We must also mention that it is possible to adapt efficient exact pattern matching algorithms to this kind of approximation. For example, in [12] adaptations of the *Tuned-Boyer-Moore*

[11] and the *Skip-Search* [3] algorithm were presented.

In this paper we present solutions for various versions of the problem of matching with gaps that was introduced in [5]. This problem is defined as follows: given a musical sequence x and a motif p, find all occurrences of p in x such that $p_i = x_{j_i}, \forall i \in \{1 \dots m\}$, where m is the length of p. Note that p occurs at position j_1 of x with a gap sequence $G = (g_1, g_2, \dots, g_{m-1})$, where $g_i = j_{i+1} - j_i - 1, \forall i \in \{1 \dots m-1\}$ and $j_1 < j_2 < \dots < j_m$. The different versions of the problem of matching with gaps are extracted by the different constraints posed on the structure of the gaps.

The organization of the paper is as follows. In Section 2 some definitions are given. In Sections 3 and 4 the problem of computing approximate occurrences with *α-bounded gaps* is considered. In Section 5 we consider the problem of computing approximate occurrences with *α-strict-bounded gaps*. Some of these algorithms were also described in [5]. In Section 6 the case with *unbounded gaps* is considered. In [5] a different algorithm was given for this version. In Section 7 we describe the case where we try to *minimize the sum of consecutive gaps*. In Section 8 we describe the case where we try to *minimize the sum of the differences of consecutive gaps*, while in Section 9 we consider the case where we try to bound the *difference of consecutive gaps*. In Section 10 a generalized version of the problem in Section 3 is given where the pattern is composed by strings of arbitrary size allowing the existence of gaps only between different strings. Finally, we give some conclusions in Section 11.

2. Definitions

Let Σ be an alphabet. A string is defined as a sequence of zero or more symbols from Σ. The empty string, that is the string with zero symbols, is denoted by e. The set of all strings over an alphabet Σ is denoted as Σ^*. A string x of length n is represented by the sequence $x_1, x_2, \dots, x_n$, where $x_i \in \Sigma$ for $1 \leq i \leq n$. We call w a substring of string x if x is of the form uwv for $u, v \in \Sigma^*$. We also say that substring w occurs at position $|u|+1$ of string x. The starting position of w in x is the position $|u|+1$ while position $|u|+|w|$ is said to be the end position of w in x. A string w is a prefix of x if x is of the form wu and is a suffix if x is of the form uw.

We define as the concatenation of two strings x and y the string xy. The concatenations of k copies of a string x is denoted by x^k. Note that self-concatenations can result in strings of exponential size. For two strings $x = x_1, x_2, \dots, x_n$ and $y = y_1, y_2, \dots, y_m$ such that $x_{n-i+1}, \dots, x_n = y_1 \dots y_i$ for some $i \geq 1$, the string $x_1, \dots, x_n, y_i, \dots, y_m$ is the superposition of x and y. In this case we say that x and y overlap.

At this point, we are going to give formally the notion of error introduced in approximate string matching. Assume that δ and γ are integers. Two symbols a, b of alphabet Σ are said to be *δ-approximate*, denoted as $a =_\delta b$, if and only if $|a - b| \leq \delta$. We say that two strings x, y are *δ-approximate*,

denoted as $x \stackrel{\delta}{=} y$ if and only if $|x| = |y|$ and $\forall i, x_i =_\delta y_i$.

Two strings x, y are said to be γ-*approximate*, denoted as $x =_\gamma y$, if and only if $|x| = |y|$ and $\sum_{i=1}^{|x|} |x_i - y_i| \leq \gamma$. Furthermore, we say that two strings x, y are (δ, γ)-*approximate* if both conditions are satisfied.

The error in the first case (δ-approximate) is defined locally for each symbol in a string. In the second case (γ-approximate) the error is defined in a more global sense and allows us to distribute the error on the symbols unevenly.

3. δ-Occurrence with α-Bounded Gaps

The problem of computing δ-occurrence with α-*bounded gaps* is formally defined as follows: given a string $t = t_1, \ldots, t_n$, a pattern $p = p_1, \ldots, p_m$ and integers α, δ, check whether there is a δ-occurrence of p in t with gaps whose sizes are bounded by constant α.

The basic idea of the algorithm described in [5] is the computation of continuously increasing prefixes of pattern p in text t so that finally we compute the δ-occurrence of the whole pattern p. That is, the algorithm is an incremental procedure that is based on dynamic programming.

Define the set of all non-empty prefixes of pattern p to be Prefixes(p). Formally speaking, Prefixes(p) $= \{\pi_1, \pi_2, \ldots, \pi_m\}$, where $\pi_i = p_{1,\ldots,i}$. We denote by the set $\Delta(\pi)$ the set of positions k in text t such that there is a δ-occurrence of π with α-bounded gaps that ends at position k.

The incremental computation is involved in the construction of the following table:

$$LastOccur_j(\pi_i) = \max\{0 \leq k \leq j : (k \in \Delta(\pi_i) \text{ and } (j - k \leq \alpha)) \text{ or } (k = 0)\} \quad (1)$$

If $LastOccur_j(\pi_i) = 0$, then there are no δ-occurrences with α-bounded gaps of prefix π_i before position j in text t. Because of the fact that we expect the number of δ-occurrences with α-bounded gaps for all π_i be constant for a position j the algorithm is expected to run in linear time $O(n)$ for typical input. If $LastOccur_j(\pi_i) \neq 0$, then we have a δ-occurrence of π_i before position j. Thus, if for some j, $LastOccur_j(p) \neq 0$ then we conclude that there is a δ-occurrence of p with α-bounded gaps. The computation of this table is done incrementally on the columns of the table, that is we proceed one symbol at a time in text t and for this symbol we try to find δ-occurrences for the set Prefixes(p). Therefore, the computation of a new column of this table is implemented by trying to extend the δ-occurrences of previous prefixes by a single symbol of the text. If we succeed, then we store at the new column the new position of the δ-occurrence, otherwise we store the position of the last match (if of course the gap invariant has not been violated, in which case we store value 0).

The table described above can be obtained by using the *Dynamic Programming approach*. We fill a matrix $D_{0...m,0...n}$, where $D_{i,j}$ corresponds to $LastOccur_j(\pi_i)$. This is computed as follows:

$$D_{i,j} = \begin{cases} j & \text{if } (t_j =_\delta p_i) \text{ and } (j - D_{i-1,j-1} \leq \alpha + 1) \text{ and } (D_{i-1,j-1} > 0) \\ D_{i,j-1} & \text{if } (t_j \neq_\delta p_i) \text{ and } (j - D_{i,j-1} < \alpha + 1) \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

The above computation basically implements the incremental algorithm whose notion is described above. Thus, if $D_{i,j} = j$ then there is a match between t_j and p_i while the prefix π_{i-1} has a δ-occurrence at a position given by $D_{i-1,j-1}$ and the formed gap is $\leq \alpha$. If $D_{i,j} = D_{i,j-1}$ then there is no match between t_j and p_i, and thus we are not able to extend the δ-occurrence of prefix π_{i-1} to π_i and as a result in $D_{i,j}$ we store the previous value of $D_{i,j-1}$ as long as the gap invariant is not violated. We store in $D_{i,j}$ the value 0 in every other case. Boundary conditions of matrix D are as follow:

$$D_{0,0} = 1, D_{i,0} = 0 \text{ and } D_{0,j} = j$$

The first boundary condition is by definition while the second one is an easy consequence of the definition of table $LastOccur$. The third boundary condition is easily extracted if we assume that the empty string e is always matched with every symbol of the alphabet.

The running time of the specific algorithm is $O(mn)$ and the space used is also $O(mn)$. However, in practice we can use only linear $O(n)$ space since the computation of each row depends only on the previous row.

If we want to retrieve a match, we can use matrix D and perform a trace-back procedure. In fact the trace-back from any cell is the reversed procedure of the construction of matrix D. The time complexity of this algorithm is $O(m)$ and the space complexity is also $O(mn)$. Another option would be to use *Hirschberg's divide and conquer technique* [10]. The total time used by this approach is $O(mn)$ while the space is linear $O(m + n)$.

4. (δ, γ)-Occurrence with α-Bounded Gaps

In this section, we are going to explore the problem of computing (δ, γ)-occurrences of a pattern with α-bounded gaps. This problem is formally defined as follows: given a string $t = t_1, \ldots, t_n$, a pattern $p = p_1, \ldots, p_m$ and integers α, δ, γ, check if there is a (δ, γ)-occurrence of p with α-bounded gaps in t.

We will follow an approach similar to the approach used in the previous section. Note that in contrast to the previous problem, we have to satisfy a global criterion (γ-occurrence) simultaneously with a local criterion (α-bounded-gaps). The local criterion is satisfied by forming each π_i from the combination of the δ-occurrence of p_i with the closest to the left δ-occurrence

Fig. 1: *This figure depicts a part of matrix D, where rectangles represent its cells. The list corresponding to p_{j+1} is depicted when processing symbol t_i of the input string. The occurrences of p_j are shaded and their position is stored in this list only when the gap invariant is not violated.*

of p_{i-1}. This means that there may be some other p_{i-1} that occurred earlier without violating the gap invariant but it is not reported as an occurrence of p. As a result, we cannot maintain the global invariant posed by γ by simply throwing away occurrences of p that do not satisfy it because there are occurrences of p that are not computed by matrix D. Thus, we need to compute matrix D in a way such that occurrences of p also satisfy the global invariant.

The computation of D is performed exactly as above except that as we scan each symbol of the text (that is, as we complete each column of matrix D) we maintain for each p_i a double linked list with occurrences of π_{i-1}, such that the gap invariant is not violated. This means that the head of the list contains the position of occurrence of p_{i-1} that is closer to the position we are currently scanning while the tail contains the occurrence of p_{i-1} that has the largest distance from the current position among the elements in the list. All occurrences stored in this list satisfy the gap invariant with respect to the current position. When we find a δ-occurrence of p_i that extends an occurrence of π_{i-1} to an occurrence of a π_i we add it to the head of the list of symbol p_{i+1}. When we scan a text symbol we may also delete the element at the tail of the double linked list since the gap invariant may be violated. Note that in every symbol scanned we make $O(1)$ work in each list since we may delete or insert at most one element. Note also that the maximum length of such a list will be equal to α.

The use of the lists is as follows (see Fig. 1). When we encounter a δ-occurrence of p_i we form π_i by letting p_{i-1} be the minimum error δ-occurrence of p_{i-1} among the δ-occurrences stored in the list corresponding to p_i. In order to find the minimum error δ-occurrence we have to scan the whole list that results in a time cost of $O(\alpha)$. We can reduce this cost by using *min-queues*. A min-queue is a data structure maintaining a list of elements with each element having each own key under the following operations: *makequeue(e)* (create and return a new queue with only one element e), *push(e)* (insert e as the new first element, the previous i-th element becomes the $(i+1)$-th), *eject()* (return an ordered pair of the last element of the queue and the queue containing the first through the second-to-last element of the queue), *findmin()* (find and return an element of minimum key in the queue). In [9] it was shown how to build min-queues that support each of the aforementioned operation in worst case constant time.

We also keep the costs of occurrences in an auxiliary matrix C. This matrix is constructed simultaneously with matrix D. When $D(i,j) = 0$ then the corresponding $C(i,j)$ contains the cost of the occurrence of π_i at position j of the text. The cost is the sum of all δ-errors introduced in each symbol

1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	1	0	0	0	6	6	0	0	10	10	11	0	0	15
0	0	2	2	0	0	0	7	8	8	0	11	12	12	0	0
0	0	0	3	4	0	0	0	0	9	0	0	0	13	0	0

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	1	0	0	0	0	0	0	0	0	0	1	0	0	1
0	0	1	1	0	0	0	1	0	0	0	1	0	0	0	0
0	0	0	1	2	0	0	0	0	0	0	0	0	0	0	0

Fig. 2: Assuming a text $t = bceddadceabcecb$ ($n = 15$), a pattern $p = ace$ ($m = 3$), $\alpha = 1$, $\delta = 1$ and $\gamma = 1$ we depict the two matrices D (the upper matrix) and C (the lower matrix). This example is very simple but exhibits the basic steps of the algorithm for finding (δ, γ)-occurrences with α-bounded gaps. Thus, in $D_{3,8} = 7$ we have an occurrence of prefix $\pi_2 = ac$ whose total error is $C_{2,7} = 1$ since c is by one different than d. However, in $D_{3,9} = 8$ we change to a new occurrence since the total error is minimized to zero ($C_{2,8} = 0$). Finally, occurrences 3, 9 and 13 are (δ, γ)-occurrences since occurence 4 has total error equal to 2 which violates the fact that $\gamma = 1$.

of the occurrence of prefixes. When $D(i, j) \neq 0$ then $C(i, j)$ gives the total error of this ocurrence. In this way, if there is an occurrence at matrix D (row m of matrix D) then by using matrix C we can deduce whether this is a γ-occurrence as well or not. In Fig. 2 an example of the above procedure is depicted. By using min-queues the time complexity of the problem is $O(nm)$ while the space complexity becomes $O(nm + m\alpha) = O(nm)$.

5. δ-Occurrence and (δ, γ)-Occurrence with Strict Bounded Gaps

The problem of computing δ-occurrence with *α-strict-bounded gaps* is formally defined as follows: given a string $t = t_1, \ldots, t_n$, a pattern $p = p_1, \ldots, p_m$ and integers α, δ, check if there is a δ-occurrence of p with α-strict-bounded gaps in t, that is with gaps with length equal to α. Obviously, this problem is a restriction of the problem defined in Section 3. Due to the fact that gaps must have length equal to α, we can easily solve the problem by splitting t into substrings w_i as follows:

$$w_k = t_k, t_{k+\alpha+1}, t_{k+2(\alpha+1)}, \ldots, \forall k \in \{1 \ldots \alpha + 1\}$$

Using known algorithms for δ-approximate matching such as *δ-Turbo-Boyer-Moore* [12] or *Shift-And* [2] we can check whether p occurs in any of substrings w_i. If, for example, p occurs in w_i at position j, then we say that there is a δ-occurrence of p with α-strict-bounded gaps at position $i + (j - 1)(a + 1)$. It is easy to see that since $\sum_{i=1}^{\alpha+1} |w_i| = n$ the running time is bounded by the complexity of the selected algorithms.

The problem of computing (δ, γ)-occurrence with *α-strict-bounded gaps* is formally defined as follows: given a string $t = t_1, \ldots, t_n$, a pattern $p = p_1, \ldots, p_m$ and integers α, δ, γ, check if there is a (δ, γ)-occurrence of p with α-strict-bounded gaps in t. This problem can be solved similarly, but

instead of using δ-*Turbo-Boyer-Moore* or *Shift-And* algorithms we can use (δ, γ)-*Turbo-Boyer-Moore* [12] or *Shift-Plus* [2]. The running time is again bounded by the complexity of the selected algorithms.

6. δ-Occurrence and (δ, γ)-Occurrence with Unbounded Gaps

The problem of computing δ-occurrence with *unbounded gaps* is formally defined as follows: given a string $t = t_1, \ldots, t_n$, a pattern $p = p_1, \ldots, p_m$ and an integer δ, check if there is a δ-occurrence of p with unbounded gaps in t. The problem is trivial when we look for one δ-occurrence of p in text t. The idea is to search for the first δ-occurrence of each symbol of p in text t. In this way we first search for p_1, then for p_2, etc. The time complexity of this algorithm is easily proved to be linear $O(n)$.

We saw above that it was trivial to solve the δ-occurrence with unbounded gaps problem and that is because gaps are unbounded and at the same time the error definition is local and depends only on the current symbols. The above approach cannot be applied in the case of computing a (δ, γ)-occurrence with unbounded gaps. This problem is defined as follows: given a string $t = t_1, \ldots, t_n$, a pattern $p = p_1, \ldots, p_m$ and integers δ, γ, check if there is a (δ, γ)-occurrence of p with *unbounded gaps* in t.

A simple algorithm for this problem is to use the algorithm presented for the problem of computing a (δ, γ)-occurrence with α-bounded gaps setting $\alpha = +\infty$ or an integer above $n - 1$. This value of α does not complicate the algorithm, and the time complexities remain unaffected (by using min-queues). So, the time complexity of the algorithm becomes $O(mn)$ as the one described in Section 4.

7. δ-Occurrence Minimizing Total Sum of Gaps

The problem of computing a δ-occurrence of a pattern p in a text t *minimizing the total sum of gaps* is formally stated as follows: given a text string $t = t_1, \ldots, t_n$, a pattern $p = p_1, \ldots, p_m$ and an integer δ, check if there is a δ-occurrence of p with gaps minimizing the quantity $\sum_{i=1}^{m-1} g_i$. The versions described in the previous sections correspond to local constraints on the length of the gaps while the version described here corresponds to a global constraint on the length of the gaps. The problem considered here is related to the episode matching problem ([6]): find the shortest substring of the text that contains the pattern as a subsequence. There is also related work in the field of music retrieval where a local criterion for gaps is used ([7]). In the sequel we will show how a slight variation of a dynamic programming algorithm given in [6] for the episode matching problem, can be used to solve the specific problem.

The algorithm first will locate the *minimal* substrings of t containing a δ-occurence of p, and then from these substrings it will collect the one with the minimum sum of gaps. A substring of t is called *minimal*, if no proper

Fig. 3: *The structure of the graph $H = (V, E)$ is depicted.*

substring of it contains p. It follows from the above description that the time cost of the algorithm is bounded from above by the cost of computing the *minimal* substrings of t.

We can compute the *minimal* substrings of t that contains a δ-occurence of p by filling a matrix $D_{0\ldots m,0\ldots n}$, where $D_{i,j}$ is the largest value k such that $t[k \ldots i]$ contains a δ-occurrence of $p[1 \ldots j]$. Then $\forall i, j : k = D_{i,j} > D_{i-1,j}$, $t[k \ldots i]$ is a minimal substring of t for the pattern $p[1 \ldots j]$. So, for all the entries of the table satisfying $D_{i,m} > D_{i-1,m}$, $t[D_{i,m} \ldots i]$ is a minimal substring containing a δ-occurence of p.

The matrix D can be computed by dynamic programming using the following recurrence relation:

$$D_{i,j} = \begin{cases} D_{i-1,j-1} & \text{if } t_i =_\delta p_j \\ D_{i-1,j} & \text{otherwise} \end{cases} \tag{3}$$

The initial conditions for the matrix D are $D_{0,j} = 0$ for $j = 1 \ldots, m$ and $D_{i,0} = i + 1$ for $i = 0, \ldots, n$. The time complexity of the construction is bounded from above from the time to construct the matrix D which is $O(nm)$.

8. δ-Occurrence Minimizing Total Difference of Gaps

The problem of computing a δ-occurrence of a pattern p in a text t *minimizing the total difference of gaps* is formally stated as follows: given a text string $t = t_1, \ldots, t_n$, a pattern $p = p_1, \ldots, p_m$ and an integer δ, check if there is a δ-occurrence of p with gaps minimizing the quantity $\sum_{i=1}^{m-2} G_i$, where $G_i = |g_i - g_{i+1}|$.

Initially, we construct a directed acyclic graph (DAG) $H = (V, E)$ by traversing the text for each symbol $p_i, 1 \leq i \leq m$ and creating a node v_i^j, $1 \leq j \leq n$ whenever $p_i =_\delta t_j$. In this way, for each symbol p_i of pattern p we may create as many as n nodes v_i^j. As a result we will have nm nodes at most. The construction is made in such a way that the nodes are divided in layers of nodes, where each layer corresponds to the δ-occurrences of a specific symbol p_i of pattern p.

The set of edges E will be constructed as follows. Edges among nodes of the same layer are forbidden. This is because we would like the edges to represent all the different δ-occurrences of a pattern p in text t. We introduce a new directed edge between two nodes v_i^j and $v_{i'}^{j'}$ if and only if $i' = i+1$ and $j' > j$. All nodes that correspond to a δ-occurrence of the symbol p_1, that is nodes v_1^j that lie at the first layer, are connected to a node s ($s \rightarrow v_1^j$). All nodes that correspond to a δ-occurrence of the symbol p_m are connected to a node d ($v_m^j \rightarrow d$). In Fig. 3 the structure of the graph is depicted. To each

edge we assign a cost proportional to the length of the gap defined by this occurrence. In this way, the edge $e = (v_i^j, v_{i+1}^{j'})$ is given weight $w_e = j' - j$. Edges starting from s or ending at d are given zero weights. It is obvious that in the worst case set E will have size $O(n^2m)$. Concluding, we can compute set V in time $O(nm)$ while set E is computed in time $O(n^2m)$. Thus, the time complexity as well as the space complexity is asymptotically equal to $O(n^2m)$. Applying a shortest path algorithm on this graph would solve the problem of minimizing the sum of the gaps in a δ-occurrence of pattern p in text t. This would incur a time cost of $O(n^2m)$ since the graph H is a DAG. However, in order to solve the problem described at the start of the section we need to make a transformation on this graph.

From graph H we construct a new graph H'. H' is implemented by contracting two nodes v_i^j and $v_{i+1}^{j'}$ that are connected by the directed edge $e = (v_i^j, v_{i+1}^{j'})$ into a node v'_e. If the edge $f = (v_{i+1}^{j'}, v_{i+2}^{j''})$ does also exist in H (in H' is represented by the node v'_f) then in H' we introduce the edge $e' = (v'_e, v'_f)$. The weight of the edge e' in H' is defined as the difference between gap lengths corresponding to weights at edges e and f in H. The new graph H' may have as many as $O(n^2m)$ nodes and $O(n^2m)$ edges. Because of the fact that H' is a DAG (Directed Acyclic Graph) we can compute a shortest path in $O(n^2m)$ time by topologically sorting it. The computation of the shortest path in this graph provides us with the solution to this problem using $O(n^2m)$ space in $O(n^2m)$ time.

The space complexity can be reduced from $O(n^2m)$ to $O(n^2)$ since it is possible to simulate the shortest path computation during constructing H'. This is possible since during the construction of H' we need the nodes and edges that correspond to three consecutive symbols p_{i-1}, p_i and p_{i+1} of the pattern p when the current symbol scanned is p_{i+1}. Scanning through the pattern and constructing the required parts of H', one can compute the shortest path to each new node by taking the minimum over the incoming edges. This procedure would reduce the space complexity to $O(n^2)$.

9. δ-Occurrence and (δ, γ)-Occurrence with ϵ-Bounded-Difference Gaps

The problem of computing a δ-occurrence with *ϵ-bounded-difference gaps* is formally defined as follows: given a string $t = t_1, \ldots, t_n$, a pattern $p = p_1, \ldots, p_m$ and integers δ, ϵ, check if there is a δ-occurrence of p with gaps satisfying $G_i = |g_i - g_{i+1}| < \epsilon$. This means that we demand that the length of consecutive gaps differ by at most an integer ϵ.

We make use of the algorithm described in Section 8 for computing δ-occurrence minimizing total difference of gaps with some changes. First, the size of each gap is not bounded and as a result we can assume that $\alpha = +\infty$ or that $\alpha \geq n + 1$. In addition, we need to keep track of the differences of consecutive gaps. This can be accomplished by introducing a graph H' similar to that of Section 8.

This graph, as described in the previous section maintains all the information about the differences of consecutive gaps. This graph is constructed in $O(n^2m)$ time and uses $O(n^2)$ space. In order to solve the problem in hand we just have to find a path from node $s \in H'$ to node $d \in H'$ where each edge has absolute weight less that ϵ. In order to accomplish this we first scan the graph and throw away each edge with absolute weight larger than ϵ. The resulting graph H'' has edges with weight less than ϵ and as a result the problem can be solved by just finding a path from node s to node d. In order to accomplish this a DFS (Depth-First Search) scanning is enough.

The problem of (δ, γ)-occurrence with ϵ-bounded-difference gaps can be solved similarly. The only difference is that during the depth first scan we also keep track of the sum of consecutive gaps, and report only the paths from s to d that satisfy the γ criterion. It is easy to see that the time and space complexities remain the same.

10. δ-Occurrence of a Set of Strings with Bounded Gaps

Assume a set of strings $w_1, w_2, \ldots, w_m \in \Sigma^*$, where the size of each string is arbitrary. The problem of computing a δ-occurrence of a *set of strings with bounded gaps* is formally defined as follows: given a text string $t = t_1, \ldots, t_n$, a set of strings $w_1, w_2, \ldots, w_m \in \Sigma^*$ and an integer δ, check if there is a δ-occurrence of each string w_i in text t such that if $t[l_1 \ldots] =_\delta w_1, t[l_2 \ldots] =_\delta w_2, \ldots, t[l_m \ldots] =_\delta w_m$ then $g_1 = l_2 - l_1 \leq \Delta$ and generally $g_i = l_{i+1} - l_i \leq \Delta$. Note that all w_i must have an occurrence in text and that the appearance is in strictly increasing order with respect to i. Note that gaps are allowed only between strings w_i.

We only give an outline of the solution since it is quite similar to that of Section 3. Define the pattern p to be $p = w_1 w_2 \ldots w_m$. Then, construct matrix D by finding in each row i the δ-occurrence of the substring w_i exactly as we did when we tried to find the δ-occurrence of a single symbol p_i. This could be accomplished by using a simple algorithm with character by character comparison. The time complexity of such a procedure is linear as a function of the length of the pattern w_i. This procedure would incur time cost equal to $O(n(|w_1| + |w_2| + \cdots + |w_m|))$ and space $O(n(|w_1| + |w_2| + \cdots + |w_m)|)$.

11. Conclusions

In this paper we introduced new versions of approximate string matching with gaps. The innovation of the specific versions lies in the conditions posed on the gaps between appearances of consecutive pattern symbols in the text. We propose algorithms for these problems based mainly on dynamic programming and on the equivalence of some optimization problems on strings to the shortest path problem on graphs. The proposed versions of approximate string matching are applied mainly in the field of computer-aided

musical analysis. Finally, we would like to see new versions of approximate string matching with gaps as well as more efficient algorithms than the one described above.

References

[1] E. Cambouropoulos, T. Crawford and C.S. Iliopoulos. Pattern Processing in Melodic Sequences: Challenges, Caveats and Prospects. *In Proc. of the AISB'99 Convention (Artificial Intelligence and Simulation of Behavior)*, Edinburgh, U.K., 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 Proc. of the* 10^{th} *Australian Workshop on Combinatorial Algorithms*, pp. 129-144, Perth, WA, Australia, 1999.

[3] C. Charras, T. Lecroq and J. D. Pehoushek. A very fast string matching algorithm for small alphabets and long patterns. *In Proc. of the* 9^{th} *Annual Symposium on Combinatorial Pattern Matching*, No. 1448 in Lecture Notes in Computer Science (LNCS), pp. 55-64, Piscataway, NJ, 1998, Springer-Verlag, Berlin.

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

[5] Maxime Crochemore, Costas S. Iliopoulos, Yoan J. Pinzon and Wojciech Rytter. Finding Motifs with Gaps. Unpublished manuscript.

[6] Gautam Das, Rudolph Fleischer, Leszek Gasienec, Dimitris Gunopulos, and Juha Karkkainen. Episode Matching. *In Proc. of the* 8^{th} *Symposium on Combinatorial Pattern Matching (CPM'97)*, LNCS 1264, Spinger 1997, pp.12-27.

[7] Matthew Dovey. A Technique for "Regural Expression" Style Searching in Polyphonic Music. *In Proc. of the* 2_{nd} *Annual International Symposium on Music Information Retrieval (ISMIR 2001)*, pp. 179-18.

[8] V. Fischetti, G. Landau, J. Schmidt and P. Sellers. Identifying periodic occurrences of a template with applications to protein structure. *In Proc. of the* 3^{rd} *Combinatorial Pattern Matching*, Lecture Notes in Computer Science, vol. 644, pp. 111-120, 1992.

[9] H. Gajewska and R. E. Tarjan. Deques with heap order. *Information Processinf Letters*, 12:4, pp. 197-200, 1986.

[10] D.S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Comm. Assoc. Comput. Mach.*, 18:6, pp. 341-343, 1975.

[11] A. Hume and D.M. Sunday. Fast String Searching. *Software-Practice and Experience*, 21(11):1221-1248, 1991.

[12] C.S. Iliopoulos, T. Lecroq and Y.J. Pinzon. Approximate String Matching in Musical Sequences. Submitted, (2000).

[13] S. Karlin, M. Morris, G. Ghandour and M.Y. Leung. Efficient Algorithms for molecular sequences analysis. *Proc. Natl. Acad. Sci.*, USA, 85:841-845, 1988.

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

[15] A. Milosavljevic and J. Jurka. Discovering simple DNA sequences by the algorithmic significance method. *Comput. Appl. Biosci.*, 9:407-411, 1993.

[16] P.A. Pevzner and W. Feldman. Gray Code Masks for DNA Sequencing by Hybridization. *Genomics*, vol. 23, pp. 233-235, 1993.

[17] P.Y. Rolland and J.G. Ganascia. Musical Pattern Extraction and Similarity Assessment. *In Readings in Music and Artificial Intelligence*, E. Miranda (editor), Harwood Academic Publishers (forthcoming), 1999.

[18] J.P. Schmidt. All shortest paths in weighted grid graphs and its applications to finding all approximate repeats in strings. *In Proc. of the* 5^{th} *Symposium on Combinatorial Pattern Matching (CPM'94)*, Lecture Notes in Computer Science, 1994.

[19] S.S. Skiena and G. Sundaram. Reconstructing strings from substrings. *In J. Compu-*

tational Biol., vol. 2, pp. 333-353, 1995.

[20] G.A. Stephen. String Searching Algorithms. *Lecture Notes Series on Computing*, vol. 3, World Scientific Publishing, ISBN: 981-02-1829-X, 1994.

[21] E. Ukkonen. Finding approximate patterns in strings. *Journal of Algorithms*, 6:132-137, 1985.