On Minimizing Pattern Splitting in Multi-Track String Matching*

Kjell Lemström and Veli Mäkinen

Department of Computer Science, University of Helsinki
P.O. Box 26 (Teollisuuskatu 23), FIN-00014 Helsinki, Finland
{klemstro,vmakinen}@cs.helsinki.fi

Abstract. Given a pattern string $P = p_1 p_2 \cdots p_m$ and K parallel text strings $\mathbb{T} = \{T^k = t_1^k \cdots t_n^k \mid 1 \leq k \leq K\}$ over an integer alphabet Σ, our task is to find the smallest integer $\kappa > 0$ such that P can be split into κ pieces $P = P^1 \ldots P^\kappa$, where each P^i has an occurrence in some text track T^{k_i} and these partial occurrences retain the order. We study some variations of this minimum splitting problem, such as splittings with limited gaps and transposition invariance, and show how to use sparse dynamic programming to solve the variations efficiently. In particular, we show that the minimum splitting problem can be interpreted as a shortest path problem on line segments.

1 Introduction

In multi-track string matching [4, 11] the task is to find occurrences of a pattern across parallel strings. Given a pattern string $P = p_1 \cdots p_m$ and K parallel text strings $\mathbb{T} = \{T^k = t_1^k \cdots t_n^k \mid 1 \leq k \leq K\}$ over an integer alphabet Σ, there is such an occurrence at j iff $p_1 = t^{k_1}_{j-m+1}, \cdots, p_m = t_j^{k_m}$, where $k_i \in [1, K]$ for $1 \leq j \leq m$.

This basic problem is a generalization of exact string matching for multi-track texts. An efficient solution is achievable by casting the problem into one of subset matching; the recent result of Cole and Hariharan [3] gives an $O(Kn \log^2 Kn)$ time solution. One can also deploy bit-parallelism. It is easy to implement the naïve algorithm to work in time $O((Kn + mn)\lceil|\Sigma|/w\rceil)$ by generalizing the solution of Iliopoulos and Kurokawa [9] for the case $|\Sigma| \leq w$, w denoting the size of machine words in bits.

In a variation of the problem the occurrences may be transposed, i.e., in the formulation above there is an occurrence if $p_1 = t^{k_1}_{j-m+1} + c, \cdots, p_m = t_j^{k_m} + c$ for some constant c. Lemström and Tarhio [11] give an efficient bit-parallel filtering algorithm for solving the transposition invariant multi-track string matching problem. After an $O(nK\lceil|\Sigma|/w\rceil)$ preprocessing phase, the algorithm runs in time $O(n\lceil m/w\rceil + m + d)$ and space $O(n\lceil|\Sigma|/w\rceil + e)$ where d and e denote factors dependent on the size of the alphabet. The found candidates can then

* In *Proc. CPM 2003*, LNCS 2676, pp. 237–253, 2003. © Springer-Verlag

be confirmed, e.g., by a checking algorithm working in time $O(mKn)$ and space $O(m\lceil|\Sigma|/w\rceil)$ [11].

In some cases it is useful to allow *gaps* between the matching elements, i.e., $p_i = t_j^{k_i}, p_{i+1} = t_{j+1+\ell_i}^{k_{i+1}}$ for $1 \leq i \leq m-1$ with some $0 \leq \ell_i \leq n-j-1$. The gaps ℓ_i can be controlled by an integer $0 \leq \alpha \leq \infty$ by requiring $\ell_i \leq \alpha$ for all i. Limited gaps are considered, for instance, in [6, 9] while unlimited gaps ($\alpha = \infty$) in [13, 14].

In this paper, we are interested in finding the smallest integer $\kappa > 0$ such that P may be split into κ pieces $P = P^1 \ldots P^\kappa$ where each P^i has an occurrence in some text track T^{k_i} and the occurrences of the consecutive pieces must retain the order. We control the gaps between the occurrences of P^{i-1} and P^i by using the integer α.

We study this *minimum splitting problem* with its variations. Our solutions use sparse dynamic programming based on the match set $M = \{(i,j,k) \mid p_i = t_j^k\}$. The problem of unlimited gaps is solved in time $O(|M|+m+Kn)$ and space $O(|\Sigma| + |M| + m + n)$. Applying this algorithm to the transposition invariant case results in time complexity $O(mKn)$. Moreover, the result generalizes to the case of α-limited gaps. In the case where the matching symbol pairs of M form relatively long diagonal runs, called *line segments*, we obtain a more efficient algorithm solving the problem. Having constructed the set of maximal line segments $\hat{S}$ in time $O(|\Sigma|^2 + |\Sigma|m + m^2 + Kn + \hat{S}\log K)$, the minimum splitting problem is solved in time $O(|\hat{S}|\log n)$. Modifications of this algorithm find the α-limited gap occurrences in time $O(|R|\log n)$, where $|R| \leq \min(|\hat{S}|^2, |M|)$. A more efficient algorithm, running in time $O(\kappa Kn)$, is given for the case $\alpha = 0$.

The problem has a natural application in content-based music retrieval: the pattern is a short excerpt of a (monophonic) query melody that may be given, e.g. by humming. Then the melody is searched for in a multi-track string representing a music database of polyphonic (multi instrument) music. However, it is musically more pertinent not to allow the pattern to be totally distributed across the tracks, but to search for occurrences of the pattern that make as few jumps as possible between different instruments. The continuity of a "hidden melody" is considered by limiting the lengths of the gaps.

Consider, for instance, the excerpt given in Fig. 1. The excerpt can be represented by a pitch string $P = efghc$, or in a transposition invariant way, by using the pitch intervals between the notes: $P' = +1, +2, +4, +1$. Fig. 2 is an excerpt from a polyphonic, four-track vocal music by Selim Palmgren. The four tracks (denoted TI, TII, BI, and BII) are divided into two systems, both containing two tracks. The two tracks within a system can be discriminated by observing the direction of the note beams; the upper track contains notes with upper beams, the lower tracks with lower beams. The excerpt in Fig. 1 has two 0-limited ($\alpha = 0$), transposed occurrences of $\kappa = 2$ in Fig. 2. The first starts at the second note in the lowest, BII track, and is distributed across two tracks (BII and TII). The second occurrence starts at the eighth note and is distributed across tracks BII and TI. It is noteworthy, that the latter occurrence captures the transition of

the musical melody from the lowest tone to the uppermost tone (the perceived melody resides in the 10 first notes of track BII, and in the 5 last notes of TI).

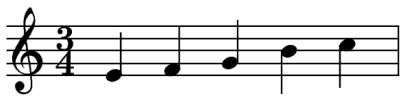

Fig. 1. A monophonic query melody.

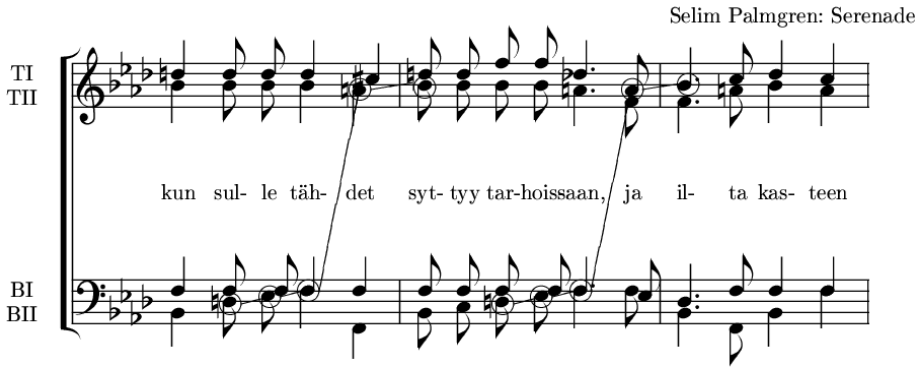

Fig. 2. An excerpt of a four-track vocal composition. The query melody in Fig. 1 has two transposed occurrences such that $\kappa = 2$ and $\alpha = 0$.

2 Definitions

Let Σ denote a fixed integer *alphabet*, i.e. $\Sigma \subset \mathbb{Z}$. The size of Σ is defined as $|\Sigma| = \max(\Sigma) - \min(\Sigma) + 1$. A *string* A over an integer alphabet Σ is a sequence $A = a_1 a_2 \cdots a_m$, where each *character* $a_i \in \Sigma$. The length of A is denoted by $|A| = m$. The string of length 0 is denoted by ϵ.

A sequence $a_i \cdots a_j = A_{i \ldots j}$ is a *substring* of A, where $1 \leq i \leq j \leq m$. Substring $A_{1 \ldots j}$ is called a *prefix* and substring $A_{j \ldots m}$ is called a *suffix* of A.

A *multi-track text* $\mathbb{T}$ is a set of equal length strings. The cardinality of $\mathbb{T}$ is denoted by $||\mathbb{T}|| = K$ and the length of each string in $\mathbb{T}$ is denoted by $|\mathbb{T}| = n$. A multi-track text $\mathbb{T}$ can be written as $\{T^k = t_1^k \cdots t_n^k \mid 1 \leq k \leq K\}$, where T^k denotes the string at track k.

Problem 1 *Given a pattern string P and a multi-track text $\mathbb{T} = \{T^k \mid 1 \leq k \leq K\}$, the* minimum splitting problem *is to find the smallest integer κ such that P can be split into κ pieces $P = P^1P^2 \cdots P^\kappa$, where each P^i occurs in some text track T^{k_i} and the consecutive pieces must retain the order, i.e., $P^i = T^{k_i}_{j_i-|P^i|+1 \ldots j_i}$, and $j_{i-1} < j_i - |P^i| + 1 \leq n - |P^\kappa| + 1$ ($j_0 = 0$) for $k_i \in [1, K]$.*

3 Solution Based on Sparse Dynamic Programming

We next describe a sparse dynamic programming algorithm for solving the minimum splitting problem. Let P, $\mathbb{T}$ be an instance of the problem. Let $M = M(P, \mathbb{T})$ be the set of matching character triples between P and each text in $\mathbb{T}$, i.e.

$$M = \{(i, j, k) \mid p_i = t^k_j, 1 \leq i \leq m, 1 \leq j \leq n, 1 \leq k \leq K\}. \quad (1)$$

Our algorithm fills (sparsely) an $m \times n \times K$ array (d_{ijk}) such that d_{ijk} stores the minimum splitting κ needed between $p_1 \cdots p_i$ and $t^{k_1}_{j'} \cdots t^{k_\kappa}_j$, where $1 \leq j' < j$, $k_i \in [1, K]$, and $k_\kappa = k$. Note that only cells $(i, j, k) \in M$ need to be filled. This sparse matrix is visualized in Fig. 3.

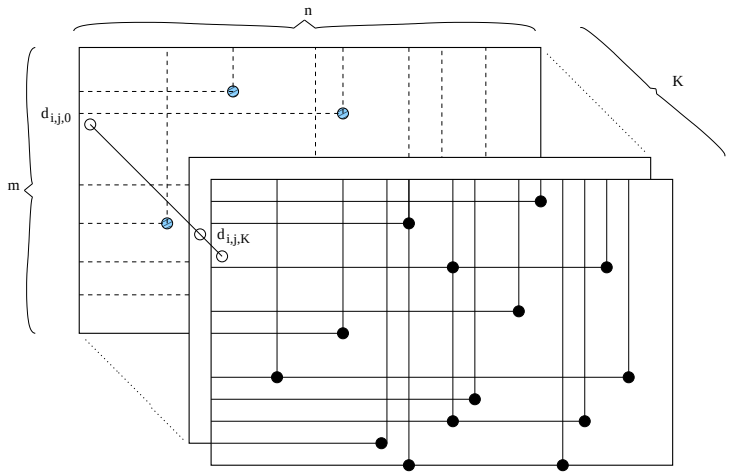

Fig. 3. The sparse matrix M.

3.1 Basic Algorithm

The recurrence for values $d_{i,j,k}$ is easily derivable from the definition. For $(i, j, k) \in M$, it is

$$d_{i,j,k} = \min \begin{cases} \text{if } (i-1, j-1, k) \in M \text{ then } d_{i-1,j-1,k} \text{ else } \infty \\ d_{i-1,j',k'} + 1 \quad \text{where } (i-1, j', k') \in M, j' < j \ (*) \end{cases}, \quad (2)$$

with the boundary condition that $d_{1,j,k} = 0$ for each $(1, j, k) \in M$. Value $\kappa = \min\{d_{m,j,k} \mid (m, j, k) \in M\}$ gives the solution to the minimum splitting problem.

An $O(mn)$ time dynamic programming algorithm is immediate from recurrence (2). We next show how to obtain $O(|M|)$ time. First, we need M constructed in some proper evaluation order: if $d_{i',j',k'}$ is needed for computing $d_{i,j,k}$, then (i', j', k') must precede (i, j, k) in M. We use the following order.

Definition 2 *The* reverse column-by-column order *of* $M \subset \{(i, j, k)\}$ *is given by sorting* M *first by* i *in ascending order, then by* j *in descending order, and then by* k *in ascending order. That is,* (i', j', k') *precedes* (i, j, k) *iff* $i' < i$ *or* $(i' = i$ *and* $(j' > j$ *or* $(j' = j'$ *and* $k' < k)))$.

Let us, for now, assume that we are given M sorted in reverse column-by-column order into an array $L(1 \ldots |M|)$, where $L(p) = (i, j, k, d)$, $1 \leq p \leq |M|$. The value d in $L(p)$ will be used to store value $d_{i,j,k}$. We denote by $L(p).\aleph$ the element $\aleph$ of $L(p)$. Moreover, let us assume that given a cell $L(p) = (i, j, k, d)$ we can access an existing cell $diag(p) = L(p') = (i-1, j-1, k, d')$ in constant time. With such representation of M, the following algorithm solves the minimum splitting problem.

Algorithm 1.
(1) $C(0) \leftarrow \infty$
(2) **for** $p \leftarrow 1$ **to** $|M|$ **do begin**
(3) $\quad L(p).d \leftarrow \min(diag(p).d, C(L(p).i - 1) + 1)$
(4) $\quad C(L(p).i) \leftarrow \min(C(L(p).i), L(p).d)$ **end**

Having run Algorithm 1, value $\kappa = \min\{L(p).d \mid L(p).i = m\}$ gives the solution.

To see that Algorithm 1 evaluates recurrence (2), consider the following. Array $C(0 \ldots m)$ is for keeping the row minima, corresponding to line 2 of recurrence (2). The reverse column-by-column evaluation guarantees that the row minimum $C(L(p).i - 1)$ is indeed taken over the correct values once computing $L(p).d$. Line 1 of recurrence (2) is taken into account in $diag(p).d$ of row (3): If $diag(p)$ does not exist, either $L(p).i = 1$ (first row) in which case we interpret $diag(p).d = 0$, or $L(p).i \neq 1$ and we interpret $diag(p).d = \infty$. This corresponds to the boundary condition of recurrence (2).

Algorithm 1 runs in time $O(|M|)$.

3.2 Constructing set M

Let us now show how to achieve an $O(|\Sigma| + m + Kn + |M|)$ total running time by constructing an array $L(1 \ldots |M|)$ with pointers $diag(p)$ to represent M as required.

Lemma 3 *Given a pattern string P of length m and a multi-track text $\mathbb{T}$ of cardinality K and of length n, the array $L(1 \ldots |M|)$ giving the reverse column-by-column order of match set $M = M(P, \mathbb{T})$ can be constructed in time $O(|\Sigma| + m + Kn + |M|)$. Also the pointers $diag(p) = L(p') = (i-1, j-1, k, d')$ from each $L(p) = (i, j, k, d)$ can be assigned within the same time bound.*

Proof. First, we construct for each character $\sigma \in \Sigma$ the list of its consecutive occurrences in $\mathbb{T}$. This is done for all characters at once by traversing through $\mathbb{T}$ in an order $t_1^1, t_1^2, \ldots, t_1^K, t_2^1, \ldots, t_n^K$. At t_j^k we insert (j, k) into the end of a list $Occ(t_j^k)$. Each insertion takes constant time, since we can use an array $Occ(1 \ldots |\Sigma|)$ to store lists. Thus, this phase is accomplished in $O(|\Sigma| + Kn)$ time.

Second, we traverse through P, and at p_i we get the ith *row* of M by copying list $Occ(p_i)$ and by adding i into each cell (j, k) to produce cell (i, j, k). At the end, we have produced M, but it is in row-by-row order. However, it is easy to produce the reverse column-by-column order; traverse through M and add each cell (i, j, k) to the beginning of a list $Col(j)$. Concatenating lists $Col(1)Col(2) \cdots Col(n)$ gives M in reverse column-by-column order. The second phase requires $O(m + |M| + n)$ time.

Finally, we assign the pointers $diag(p)$: We sort M in *diagonal-by-diagonal order*, first by values $j - i$, then by j, and then by k. This takes $O(|M|)$ time (implementation is analogous to the case considered above). Once M is ordered, the pointers are easy to assign. Consider first that $K = 1$. Then the cells to be linked are consecutive, and pointers can be assigned in constant time. When $K > 1$, one can still do this in constant time per cell by merging the sets $F(i, j) = \{(i-1, j-1, k) \mid (i-1, j-1, k) \in M\}$ and $S(i, j) = \{(i, j, k) \mid (i, j, k) \in M\}$ by value k. When $F(i, j)$ and $S(i, j)$ are merged, the cells to be linked become consecutive. Each merging can be done in linear time in the size of sets $F(i, j)$ and $S(i, j)$, since both sets can be obtained from M already in correct order. Each cell of M belongs to (at most) one $F(i, j)$ and to (at most) one $S(i, j)$, which gives the $O(|M|)$ time for assigning all $diag(p)$ pointers.

Summing all the running times, we get the claimed bound. □

3.3 Limiting Gaps with α

Let us now consider the case of α-limited gaps. We will show that the time bound $O(|\Sigma| + m + Kn + |M|)$ of the unlimited case, can be achieved in this case, as well.

When gaps are limited, the condition (*) in recurrence (2) becomes $j - \alpha \leq j' < j$. A straightforward generalization of Algorithm 1 would result in time complexity $O(|\Sigma| + m + Kn + |M|\alpha)$. We can do better by applying a queue data structure Q supporting the following operations:

- $v \leftarrow Q.Min()$: Return the minimum value v of elements in the queue;
- $Q.Add(j, v)$: Add new element j as the first element with value v;
- $j' \leftarrow Q.KeyOfLast()$: Return the key of the last element; and

– $Q.Remove()$: Remove the last element.

The modification for Algorithm 1 is evident: Instead of maintaining each row minimum in $C(i)$, use a queue $Q(i)$ for each row i. The algorithm changes into the following.

Algorithm 2.
(1) **for** $p \leftarrow 1$ **to** $|M|$ **do begin**
(2) $\quad L(p).d \leftarrow \min(diag(p).d, Q(L(p).i-1).Min() + 1)$
(3) $\quad Q(L(p).i).Add(L(p).j, L(p).d)$
(4) $\quad$ **while** $L(p).j - \alpha > Q(i).KeyOfLast()$ **do** $Q(i).Remove()$ **end**

Analogously to Algorithm 1, we interpret $diag(p).d = \infty$ if $diag(p)$ does not exist, except when $L(p).i = 1$ we interpret $diag(p).d = 0$.

If the queues are implemented as min-dequeues [7], all the above mentioned operations can be supported in constant time.[1] Thus, the algorithm runs in time $O(|\Sigma| + m + Kn + |M|)$.

3.4 Transposition Invariance.

The $O(|M|)$ algorithms are useful for the transposition invariant case, as well. Let M^c be the set of matches for transposition c, i.e. $M^c = M^c(P, \mathbb{T}) = \{(i,j,k) \mid p_i + c = t^k_j\}$. Then $\sum_{c \in \mathbb{C}} |M^c| = mKn$, where $\mathbb{C} = \{t^k_j - p_i\}$ [12]. Note that $\mathbb{C}$ is the set of relevant transpositions, since for other $c \in \mathbb{R} \setminus \mathbb{C}$, the corresponding set M^c is empty. The sets $\{M^c \mid c \in \mathbb{C}\}$ can be constructed in $O(|\Sigma| + mKn)$ time on integer alphabets [12]. By repeating the previous $O(|M|)$ algorithms for each $c \in \mathbb{C}$, we get total running time $O(|\Sigma| + mKn + \sum_{c \in \mathbb{C}} |M^c|) = O(|\Sigma| + mKn)$.

The results of Sect. 3 are summarized below.

Theorem 4 *Given a pattern string P of length m and a multi-track text $\mathbb{T}$ of cardinality K and of length n, there is an algorithm solving the minimum splitting problem (with or without the α–restriction on gaps) in $O(|\Sigma| + m + Kn + |M|)$ time on integer alphabet, where $M = \{(i,j,k) \mid p_i = t^k_j\}$. The transposition invariant case can be solved in $O(|\Sigma| + mKn)$ time.*

4 Interpretation as a Shortest Path Problem on Line Segments

In this section, we interpret the minimum splitting problem as a geometric path problem on line segments and define the minimum jump distance. For simplicity, let us assume that we have only one text; the minor modifications for the case of multiple texts are given in Sect. 4.4.

[1] Min-deques were also used by Crochemore et al. [6] for a similar problem.

The set of possible matches M is written as $M = M(P,T) = \{(i,j) \mid p_i = t_j\}$ during this section. Consider the $m \times n$ matrix, where pairs in M are sparse set of points. Diagonally consecutive points $(i,j), (i+1,j+1), \ldots, (i+\ell-1, j+\ell-1) \in M$ form a line segment S in this grid. Let us denote by $\hat{S} = \hat{S}(P,T)$ the set of all *maximal* line segments of M. A line segment $S = (i,j) \cdots (i+\ell-1, j+\ell-1)$ is maximal if $(i-1, j-1), (i+\ell, j+\ell) \notin M$. Let us denote by $start(S) = (i,j)$ and $end(S) = (i+\ell-1, j+\ell-1)$ the two end points of line segment $S \in \hat{S}$. Our problem now equals the following geometric path problem: Find a minimum cost path from row 1 to row m such that only diagonal steps are allowed when inside a line segment, otherwise only horizontal steps are allowed. To be precise, we must also take the discrete nature of the problem into account; one can jump (using horizontal steps) from segment S' to segment S only if there is $p \geq 2$ such that $(i-1, j-p) \in S'$ for some $(i,j) \in S$. The cost of a path is the number of line segments in the path -1. *Minimum jump distance* equals the minimum cost path from row 1 to row m. This interpretation as a shortest path problem is visualized in Fig. 4.

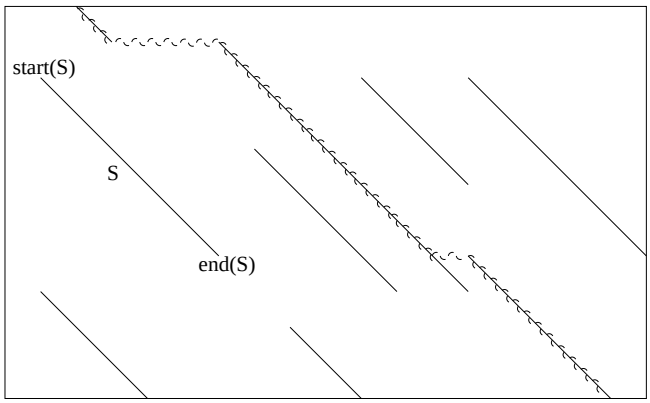

Fig. 4. Example of line segments. An optimal path from top to the bottom is visualized.

The following subsections considers, whether: (i) the set of line segments $\hat{S}$ can be constructed in a time proportional to $|\hat{S}|$, and (ii) the minimum jump distance can be computed in a similar time.

4.1 Constructing Set $\hat{S}$

Next we give an algorithm that, after $O(m^2 + |\Sigma|m + |\Sigma|^2)$ time preprocessing, constructs set $\hat{S}$ in time $O(n + |\hat{S}|)$. We use and extend some results of Ukkonen and Wood [15].

Let $Prefix(A,B)$ denote the length of the longest common prefix of A and B. Let $Maxprefix(j)$ be $\max\{Prefix(P_{i\ldots m}, T_{j\ldots n}) \mid 1 \leq i \leq m\}$, and

$H(j)$ some index i of P that gives the maximum. Let $Jump(i,j)$ denote $Prefix(P_{i\ldots m}, T_{j\ldots n})$. We have the following connection.

Lemma 5 ([15]) $Jump(i,j) = \min(Maxprefix(j), Prefix(P_{i\ldots m}, P_{H(j)\ldots m}))$.

Ukkonen and Wood show that $Maxprefix(j)$ and $H(j)$ are computable, for all j, in $O(m^2 + |\Sigma| + n)$ time by using an Aho-Corasick automaton [1] for all suffices of the pattern (or more precisely its minimized version, the suffix automaton [5]). At preprocessing, one can construct an array $Pl(1\ldots m, 1\ldots m)$ in $O(m^2)$ time such that $Pl(i',i) = Prefix(P_{i'\ldots m}, P_{i\ldots m})$ [10, 8]. From Lemma 5 it follows that, after $O(m^2 + |\Sigma| + n)$ time preprocessing, $Jump(i,j)$ can be supported in constant time for all pairs i,j.

Therefore, if we manage to call $Jump(i,j)$ only at points $(i,j) = start(S)$ for $S \in \hat{S}$, we will be able to construct $\hat{S}$ in $O(|\hat{S}|)$ time. Note that a point (i,j) equals $start(S)$ for some $S \in \hat{S}$ iff $p_{i-1} \neq t_{j-1}$ and $p_i = t_j$. We construct a data structure $\mathcal{L}$ that supports an operation $\mathcal{L}.List(x,y)$ giving the list of positions $\{i\}$ in P, where $p_{i-1} \neq x$ and $p_i = y$. Assuming that such data structure exists, the following algorithm constructs set $\hat{S}$ (represented as pairs of end points) in time $O(n + |\hat{S}|)$.

Algorithm 3.
(1) /* construct $\mathcal{L}$ */
(2) $\hat{S} \leftarrow \emptyset$
(3) **for** $j \leftarrow 1$ to n **do begin**
(4) $\quad L \leftarrow \mathcal{L}.List(t_{j-1}, t_j)$ /* $t_0 = \epsilon$ */
(5) $\quad$ **for** $k \leftarrow 1$ to $|L|$ **do begin**
(6) $\quad\quad \ell \leftarrow Jump(L(k), j)$
(7) $\quad\quad \hat{S} \leftarrow \hat{S} + ((L(k), j), (L(k) + \ell - 1, j + \ell - 1))$ **end end**

Let us now explain how to construct $\mathcal{L}$. We build a search tree of depth two representing all choices to read two characters. Let xy be two characters leading to leaf l. We associate a list of pattern positions $\{i\}$ to l such that $p_{i-1} \neq x$ and $p_i = y$. Then $\mathcal{L}.List(x,y)$ is found in linear time in the size of the output by traversing the path xy (to l) and printing the list associated with l. Now, the size of this structure is $O(|\Sigma|^2 + |\Sigma|m)$, since there are $O(|\Sigma|^2)$ leaves and each position i of P is stored in exactly $|\Sigma| - 1$ lists (except position 1 which is stored in $|\Sigma|$ lists, since we assume $p_0 \neq x$ for all x). The naïve algorithm to construct the structure (which inserts each i from $i = 1$ to $i = m$ into $\mathcal{L}$) runs in time proportional to the size of the structure.

Lemma 6 *Given a pattern string P and a text T of lengths m and n, respectively, Algorithm 3 constructs the set of maximal line segments $\hat{S}(P,T)$ in $O(m^2 + |\Sigma|m + |\Sigma|^2 + n + |\hat{S}|)$ time. Set $\hat{S}$ can be obtained in diagonal-by-diagonal order by a small adjustment of the algorithm.*

Proof. The result should be clear from the above discussion, expect for the diagonal-by-diagonal order. At line (7) of Algorithm 3, we can add the new line segment into the end of a list $D(j - L(k))$. Finally, we can catenate all the lists $D(-m), \ldots, D(n)$ to get $\hat{S}$ in diagonal-by-diagonal order. □

4.2 Computing Minimum Jump Distance

Let $d((i,j))$ give the minimum jump distance from row 1 to (i,j). To get a bound close to $O(|\hat{S}|)$, we need an algorithm doing computation only at end points of line segments. To this end, we have the following lemma.

Lemma 7 *The minimum jump distance $d(end(S))$ equals to the minimum jump distance $d(start(S))$, for each $S \in \hat{S}$.*

Proof. If this is not true, then there must be a point $(i'', j'') \neq start(S)$ in S and a point $(i''-1, j''-p'')$ in some $S'' \in \hat{S}$ (where $p'' \geq 2$) such that $d((i''-1, j''-p'')) < d(start(S)) - 1$.

On the other hand, there must be a point $(i'-1, j'-p')$ in some $S' \in \hat{S}$ such that $start(S) = (i', j')$, $p' \geq p''$, and an optimal path from row 1 to point $(i''-1, j''-p'')$ traverses through $(i'-1, j'-p')$. From which follows that $d((i'-1, j'-p')) \leq d((i''-1, j''-p''))$.

Thus, $d((i', j')) \leq d((i'-1, j'-p')) + 1 \leq d((i''-1, j''-p'')) + 1$, which contradicts the counter-argument, and the lemma follows. □

As a consequence of Lemma 7, we also know that $d((i'', j'')) = d(start(S))$ for each $(i'', j'') \in S$. Thus, it is enough to consider only end points of line segments, which are traversed e.g. in row-by-row order. Let $d(S)$ denote the minimum jump distance on line segment S. A data structure $\mathcal{T}$ stores the diagonal number $j-i$ of each line segment S together with $d(S)$. Structure $\mathcal{T}$ is updated only when some line segment starts or ends. At any point, $\mathcal{T}$ contains only those line segments that intersect row $i-1$.

Consider now that we are computing value $d((i,j))$ where $(i,j) = start(S)$ for some $S \in \hat{S}$. We need to query $\mathcal{T}.Min([-\infty, j-i))$, which gives us the value $\min\{d((i',j')) \mid S' \in \hat{S}, start(S') = (i',j'), S' \cap ([i-1] \times [0,n]) \neq \emptyset, j'-i' < j-i\}$. Then $d((i,j)) = \mathcal{T}.Min([-\infty, j-i)) + 1$. These queries can be supported by a binary search tree: use $j-i$ as the sort key, store values $d(S)$ in leaves, and maintain subtree minima in each internal node. A range query for the minimum value in a range $[l,r]$ can then be easily supported in $O(\log n)$ time. The update operations can be supported, too: When inserting a line segment, the tree can be rebalanced and the subtree minima need only be updated in the path from the new leaf to the root. Same applies for deleting a line segment. Each query takes $O(\log n)$ time, since there can be only $O(n)$ active line segments at a time.

The order of queries on $\mathcal{T}$ with respect to segment insertions and deletions is important; we must delay deletions at row $i-1$ until all queries at row i are done, and insertions at row i until all queries in that row are done. As an answer to question (ii), we conclude that the minimum jump distance can be computed

in time $O(|\hat{S}| \log n)$, once the end points of the line segments are first given in row-by-row order. By Lemma 6, we get $\hat{S}$ in diagonal-by-diagonal order. It is easy to convert $\hat{S}$ into row-by-row order in $O(\hat{S})$ time using similar techniques as in the proof of Lemma 3.

To clarify the above steps, we now give the algorithm in more detail. Let us first fix the operations described above on $\mathcal{T}$:

- $v \leftarrow \mathcal{T}.Min(I)$: Return the minimum value v of elements having keys in range I;
- $\mathcal{T}.Insert(e, k, v)$: Insert new element e with key k and value v; and
- $\mathcal{T}.Delete(k)$: Remove the element with key k.

Let us then partition the end points of line segments in $\hat{S}$ by row number and the type of end point; lists $B(i)$ and $E(i)$ contain all start and end points of line segments starting/ending at row i, respectively. Each list $B(i)$, $E(i)$, $1 \ldots m$, is in increasing order of column numbers j. Each element s of $B(i)$ (or $E(i)$) contains values (i, j, seg), where seg is a pointer to the line segment S whose end point (i, j) is. Each line segment S is associated with value $S.d$ that is used for storing the minimum jump distance from row 1 to S. The algorithm is given below.

Algorithm 4.

```
(1)  for each s ∈ B(1) do begin /* initialize first row to zero */
(2)      s.seg.d ← 0; T.Insert(s, s.j − s.i, s.seg.d); end
(3)  for i ← 2 to m do begin /* compute row-by-row */
(4)      for each s ∈ B(i) do /* compute values at row i */
(5)          s.seg.d ← T.Min([−∞, s.j − s.i)) + 1
(6)      for each s ∈ E(i − 1) do /* remove those ending at row i − 1 */
(7)          T.Delete(s.j − s.i)
(8)      for each s ∈ B(i) do /* add new line segments starting at row i */
(9)          T.Insert(s, s.j − s.i, s.seg.d); end
```

Having executed Algorithm 4, the minimum jump distance d from row 1 to row m is $d = \mathcal{T}.Min([-\infty, \infty])$.

Summing up the preprocessing time for constructing $\hat{S}$ and the computation time of Algorithm 4, we get the following result.

Theorem 8 *Given a pattern string P and text T of lengths m and n, respectively, the minimum splitting problem can be solved in time $O(|\Sigma|^2 + |\Sigma|m + m^2 + n + |\hat{S}| \log n)$, where $\hat{S}$ is the set of maximal line segments formed by the diagonally consecutive pairs of $M = \{(i, j) \mid p_i = t_j\}$.*

4.3 Limiting Gaps with α

Note that Lemma 7 does not hold if the lengths of the gaps are limited by a constant α. To see this, consider situation where there are no line segments

within α distance from $start(S)$. In this case $d(start(S)) = \infty$. Nevertheless, there can be line segments within α distance from some internal points of S, in which case $d(end(S)) < d(start(S))$ may hold.

We give a weaker lemma stating that there is a sparse set of points R, $R \subseteq M$, such that similar computation as with $\hat{S}$ is possible. The size of R is bounded by $\min(|\hat{S}|^2, |M|)$. The intuition of the lemma is, that at each line segment S it is enough to consider only points s, such that, there is an optimal path that traverses through a start point of a line segment S' and that leads to s. Moreover, the part of the path from S' to S is of a certain, fixed form.

To state the lemma, we need to introduce some concepts. The feasible paths in the sparse set M can be represented in $\hat{S}$ as follows; a *feasible path* in $\hat{S}$ is a sequence of line segments visited by the path, and a list of entry and exit points on them. Let us now define the fixed form of path mentioned above. An *α-straight path Π* from $start(S')$ to S ($S', S \in \hat{S}$) is a sequence $\Pi = \pi_1 \cdots \pi_p$ of line segments of $\hat{S}$ such that $\pi_1 = S'$, $\pi_p = S$. Path Π must satisfy the following conditions:

(i) the gaps between consecutive line segments of Π are limited by α, i.e. $j_q - j_{q-1} - 1 \leq \alpha$, where $1 < q \leq p$ and $start(\pi_q) = (i_q, j_q)$;
(ii) only a single point in each line segment is visited, i.e. point $(i_1 + q - 1, j_q + q + i_1 - i_q - 1)$ must be inside line segment π_q, where $start(\pi_q) = (i_q, j_q)$, $1 \leq q \leq p$.

Definition 1. *An* α-greedy path *is an α-straight path Π whose line segments are picked greedily maximizing the length of gaps under constraints (i-ii).*

An α-greedy path is visualized in Fig. 5.

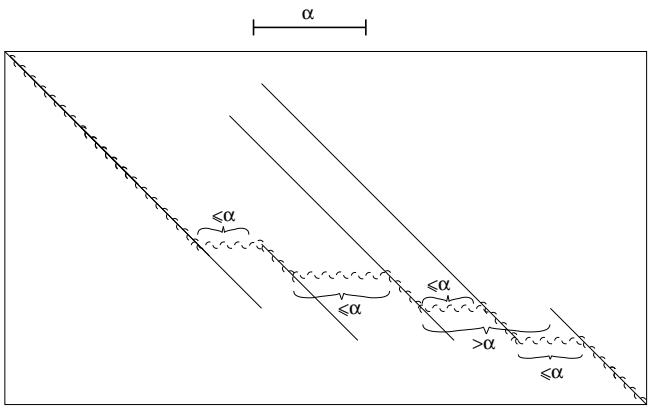

Fig. 5. An optimal path from top to the bottom. The path contains an α-greedy part from the start of the second line segment to the point where it enters the last line segment.

Lemma 9 *Let $s = (i, j)$ be a point on line segment $S \in \hat{S}$. Then there is an optimal path from row 1 to point s such that the last part of the path is an α-greedy path starting from point $start(S')$ of some line segment $S' \in \hat{S}$ to $(i - c, j - c) \in S$, $c \geq 0$.*

Proof. We will first prove that there is an optimal path that has an α-straight path as a suffix, and then we show that each α-straight path can be replaced by an α-greedy path.

Let S' be the last line segment whose start point $start(S')$ is on an optimal path from row 1 to point s. Such line segment always exists. Let us consider the last part of this path from $start(S')$ to s. We can straighten this path by removing one diagonal movement from each line segment it visits. Notice that the path is still feasible since the gaps between consecutive line segments do not change. We can continue straightening without changing the cost of the path until we meet the first start point of some line segment S'' or until the path is optimally straight, i.e. it is an α-straight path. In the latter case we are done. In the first case, we can start the same process from S'' and then by induction we finally find some line segment S''' which has an α-straight path from $start(S''')$ to point $(i - c, j - c) \in S$. It might happen that line segment S is the one that limits the straightening process at some point. This is a special case, since then we have a zero length α-straight path from $start(S)$ to $start(S)$.

Now, we still need to show that we can safely replace an α-straight path with an α-greedy path. Let $X = x_1 x_2 \cdots$ be the sequence of line segments visited by an α-straight path and $Y = y_1 y_2 \cdots$ the sequence visited by an α-greedy path such that both sequences start and end with the same line segments. If we replace x_1 with y_1, x_2 can still be reached from y_1, or x_2 is a predecessor of y_1 in the diagonal order. In the first case, we can continue on replacing x_2 with y_2. In the second case, we can omit x_2 since x_3 can be reached from y_1, or x_3 is a predecessor of y_1. Iterating this replacement for the whole X we get an equal or cheaper cost path. □

An algorithm computing the minimum jump distance under the α constraint follows from Lemma 9. In addition to start and end points of line segments, some computation on the *intersection set* R, that contains points at the intersection of $\hat{S}$ and the α-greedy paths, is needed. Set R can be computed on-the-fly by extending Algorithm 4 of previous subsection.

We now describe the required modifications to Algorithm 4. Recall data structure $\mathcal{T}$ and lists $B(i)$, $E(i)$ from Section 4.2. We need two additional operations on $\mathcal{T}$:

- $\mathcal{T}.Update(e, k, v)$: Update the value of element having key k to v (or $Insert(e, k, v)$ if key k is not found).
- $L = \mathcal{T}.List(I)$: Return the list of elements having keys in range I.

Update-operation can be supported in time $O(\log n)$ just like *Insert*. *List*-operation can be supported in time $O(\log n + occ)$, where occ is the number of elements returned. The latter operation is used for listing all line segments

in α distance from the current start point (to update values in the α-greedy paths). We cannot, however, use this operation as such at each start point, since then we could report the same intersection points multiple times. To obtain total running time $O(\min(|S|^2, |M|) \log n)$, we must ensure that we report each intersection point only once. This is done by maintaining for each row i a list $I(i-1)$ of distinct ranges, where we need to call the *List*-operation. After computing the values at row $i-1$, and collected list $I(i-1)$, we query each range of $I(i-1)$ from $\mathcal{T}$. We get an increasing list A of line segments. We can merge this list with $B(i)$ (in linear time in their sizes) to get the list F of line segments whose values need to be updated at row i. To proceed, we need to remove from A those line segments that do not continue any α-greedy path (to ensure $|S|^2$) bound). This is done by a simultaneous scan over A and over the points that formed $I(i-1)$. We get list G of line segments that continue α-greedy paths. By merging G and $B(i)$ we get a list U of line segments that continue or start new α-greedy paths. From U, we get list $I(i)$ of ranges that need to be updated at row $i+1$. We have now described in an informal level one step of the algorithm.

The detailed algorithm is given below. We need two instances of $\mathcal{T}$ since we need to be able to query two rows, $i-1$ and i, at the same time. We only give the update operations for $\mathcal{T}$ corresponding to row $i-1$. Structure $\mathcal{T}'$ for row i can be maintained similarly; at any point it contains all line segments that intersect row i, but do not start at that row. We assume that all non-initialized values are set to ∞.

Algorithm 5. /* Update operations for $\mathcal{T}'$ are not given */
(1) **for each** $s \in B(1)$ **do begin** /* initialize first row to zero */
(2) $s.seg.d \leftarrow 0$; $\mathcal{T}.Insert(s, s.j - s.i, s.seg.d)$; **end**
(3) $U \leftarrow B(1)$
(4) **for** $i \leftarrow 2$ to m **do begin** /* compute row-by-row */
(5) $I(i-1) \leftarrow GetRanges(U, \alpha)$; $A \leftarrow \emptyset$
(6) **for each** range r in $I(i-1)$ **do** $A \leftarrow Concatenate(A, \mathcal{T}'.List(r))$
(7) $F \leftarrow Merge(A, B(i))$ /* line segments to be updated */
(8) **for each** $s \in F$ **do** /* compute values at row i */
(9) $s.seg.d \leftarrow \mathcal{T}.Min([s.j - s.i - \alpha, s.j - s.i)) + 1$
(10) **for each** $s \in E(i-1)$ **do** /* remove those ending at row $i-1$ */
(11) $\mathcal{T}.Delete(s.j - s.i)$
(12) **for each** $s \in F$ **do** /* update values at row i */
(13) $\mathcal{T}.Update(s, s.j - s.i, s.seg.d)$
(14) $G \leftarrow Reduce(A, U)$ /* remove those not in α-greedy paths */
(15) $U \leftarrow Merge(G, B(i))$; /* get the set of active line segments */ **end**

Since the implementations of $Concatenate()$ and $Merge()$ are straightforward, we only describe how to implement $GetRanges()$ and $Reduce()$. In $GetRanges(U, \alpha)$, we scan over U and report each range once it is ready; Let $[l, r]$ be the current unfinished range and $s \in U$ the next segment to be processed.

If $s.j - s.i \leq r$ then $[l, r]$ becomes $[l, s_j - s_i + \alpha]$. Otherwise, $[l, r]$ is reported and a new range $[s_j - s_i + 1, s_j - s_i + \alpha]$ becomes the new unfinished range. In $Reduce(A, U)$, the logics is the following; For each $s' \in U$ there is at most one $s \in A$ such that s can be reached from s' with a jump $\leq \alpha$ and the jump is maximal among all segments that can be reached from s' with jump $\leq \alpha$ (thus, s continues an α-greedy path). With a simultaneous scan over U and A we can find *all* the segments of A that are *not maximal* for *any* $s' \in U$. This is, because once a maximal segment s is found for some $s' \in U$, we can continue with the next segment in U. Then by scanning A starting from s, we can remove all segments in between until we found the new maximal segment. Hence, both $GetRanges()$ and $Reduce()$ can be implemented in linear time in the total lengths of the input lists. Same applies for $Merge()$. Each call to function $Concatenate()$ takes constant time.

The correctness of Algorithm 5 follows from the fact that the minimum jump distance to each line segment is updated when the line segment starts, and at the intersection points of α-greedy paths.

Due to Lemma 9, these are the only points where updating is needed. The running time $O(\min(|S|^2, |M|) \log n)$ follows from two facts: (1) Each of the $|S|$ α-greedy paths can intersect at most $|S| - 1$ line segments. Algorithm 5 (due to lines 14-15) only does computation on the intersections of α-greedy paths and line segments (those intersection points that are removed in line 14 are end points of some α-greedy paths, so the computation on them in lines 5-9 is not only allowed but necessary). (2) Not depending on how many α-greedy paths intersect a point in M, each such point is accessed constant number of times (due to lines 5-7). The $\log n$ factor is from queries on $\mathcal{T}$ and $\mathcal{T}'$.

The following theorem summarizes the result.

Theorem 10 *Given a pattern string P and text string T of lengths m and n, respectively, the minimum splitting problem with α-limited gaps can be solved in time $O(|\Sigma|^2 + |\Sigma|m + m^2 + n + |R| \log n)$, $|R| \leq \min(|\hat{S}|^2, |M|)$, where $\hat{S}$ is the set of line segments formed by the diagonally consecutive pairs of $M = \{(i, j) \mid p_i = t_j\}$.*

4.4 Handling Multiple Texts

The algorithms above work with minor modifications also when $K > 1$; the preprocessing now takes $O(|\Sigma|^2 + |\Sigma|m + m^2 + Kn + |\hat{S}| \log K)$ time. The extra $\log K$ factor comes from the fact that we need to merge the sets of line segments, which are constructed for each text T^k separately. That is, we have to merge K lists that are already in row-by-row order. This takes $O(|\hat{S}| \log K)$ time. The algorithm itself takes $O(|\hat{S}| \log n)$ time or $O(|R| \log n)$ time if gaps are restricted with α. Here $\hat{S}$ is the set of (possibly parallel) line segments corresponding to the diagonally consecutive points in the (i, j) plane at fixed k in the sparse matrix $M = \{(i, j, k) \mid p_i = t_j^k\}$. Set R is the intersection set containing points at the intersection of $\hat{S}$ and the α-greedy paths.

Basically, the only modification is that the query range is changed from $j' - i' < j - i$ to $j' - i' \leq j - i$ in the algorithms (to allow jumps between parallel line segments). Also, the data structure $\mathcal{T}$ must be modified so that the key associated with a line segment is not simply $j - i$, but a combination of $j - i$ and k; the value of k can be ignored in range queries, but it must be used for separating parallel line segments.

5 Allowing no Gaps

We finally consider the special case where gaps are not allowed, i.e. $\alpha = 0$. The following fact is easy to see.

Fact 11 *Let there be a splitting of the pattern into κ pieces, starting at position j of the multi-track text $\mathbb{T}$, without gaps between the consecutive pattern piece occurrences. Then there is an equally good occurrence that can be found by the following greedy algorithm: Select the text T^k whose jth suffix has the longest common prefix with the pattern. Let this common prefix have length l. Iterate the same algorithm from position $j + l$ with pattern suffix P_{l+1}, until a splitting to κ pieces is found.*

The above greedy algorithm is given more formally below.

Algorithm 6.
(1) **for** $j = 1$ to $n - m + 1$ **do begin**
(2) $(k, i, l) \leftarrow (1, 1, 0)$
(3) **while** $k \neq 0$ **do begin**
(4) $(l', k) \leftarrow JumpK(i + l, j + l); l \leftarrow l + l'$
(5) **if** $l == m$ /* An occurrence found */ **end end**

Function $JumpK(i, j)$, in Algorithm 6, returns pair (l, k) such that the jth suffix of text track T^k has the longest common prefix (of length l) with the pattern suffix $P_{i...m}$. If no common prefixes are found, it returns $(0, 0)$.

We can again use the result of Ukkonen and Wood [15] (see Sect. 4.1), and support $JumpK(i, j)$ in $O(K)$ time (by computing values $Maxprefix(j)$ for each text separately). Given a threshold κ, we can limit the while-loop at lines (3–5) of Algorithm 6 so that at most $\kappa + 1$ calls to function $JumpK(i, j)$ are made at each position j. This gives an $O(m^2 + |\Sigma| + \kappa K n)$ time algorithm ($O(m^2 + |\Sigma| + Kn)$ comes from the preprocessing for $JumpK(i, j)$ queries).

Theorem 12 *Given a pattern string $P = p_1 p_2 \cdots p_m$ and a multi-track text $\mathbb{T} = \{T^k = t_1^k \cdots t_n^k \mid 1 \leq k \leq K\}$, the minimum splitting problem with $\alpha = 0$ can be solved in $O(m^2 + |\Sigma| + \kappa K n)$ time, where κ is a given threshold.*

6 Conclusions

We have studied the minimum splitting problem of string patterns in multi-track texts, and introduced efficient solutions to certain variations of the problem. Our solutions based on sparse dynamic programming (given in Sect. 3), have a direct application in music retrieval. The algorithm summarized in Theorem 4 has been implemented and included in our C-BRAHMS music retrieval engine [2].

Interpreting the problem as a shortest path problem on line segments seems to give even more efficient algorithms. For random strings the line segments are rather short, and the algorithms based on line segments are not practical. In music, where repetitions are common, the line segment based algorithms are expected to perform somewhat better. Nevertheless, we expect these algorithms to be more theoretically than practically interesting.

However, a simple modification to the problem statement makes the line segment based algorithms practically interesting. Suppose that we are given a limit γ to the minimum length of the pieces the pattern can be split into. Our preprocessing can be modified to construct a set $\hat{S}^{\gamma}$ of maximal line segments of length at least γ in $O(m^2 \log m + Kn \log m + \gamma Kn + |\hat{S}^{\gamma}| \log K)$ time. This already is practical, since set $\hat{S}^{\gamma=2}$ is expected to be much smaller than $\hat{S}^{\gamma=1}$. We leave for further work to improve upon the γKn term.

References

1. A.V. Aho and M.J. Corasick. Efficient string matching. *Communications of the ACM*, 18(6):333–340, 1975.
2. C-BRAHMS. http://www.cs.helsinki.fi/group/cbrahms/demoengine/.
3. R. Cole and R. Hariharan. Verifying candidate matches in sparse and wildcard matching. In *Proc. Symposium on Theory of Computing (STOC'2002)*, pages 592–601. ACM Press, 2002.
4. T. Crawford, C.S. Iliopoulos, and R. Raman. String matching techniques for musical similarity and melodic recognition. *Computing in Musicology*, 11:71–100, 1998.
5. M. Crochemore. String matching with constraints. In *MFCS*, pages 44–58, 1988.
6. M. Crochemore, C.S. Iliopoulos, C. Makris, W. Rytter, A. Tsakalidis, and K. Tsichlas. Approximate string matching with gaps. *Nordic Journal of Computing*, 9(1):54–65, 2002.
7. H. Gajewska and R. Tarjan. Deques with heap order. *Information Processing Letters*, 12(4):197–200, 1986.
8. Z. Galil and K. Park. An improved algorithm for approximate string matching. *SIAM J. Comput.*, 19:989–999, 1990.
9. C.S. Iliopoulos and M. Kurokawa. String matching with gaps for musical melodic recognition. In *Proc. Prague Stringology Conference 2002 (PSC'2002)*, pages 55–64, Prague, 2002.
10. G.M. Landau and U. Vishkin. Fast string matching with k differences. *Journal of Computers and Systems*, 37:63–78, 1988.
11. K. Lemström and J. Tarhio. Transposition invariant pattern matching for multi-track strings. 2003. (submitted).

12. V. Mäkinen, G. Navarro, and E. Ukkonen. Algorithms for transposition invariant string matching. In *Proceedings of the 20th International Symposium on Theoretical Aspects of Computer Science (STACS'2003)*, volume 2607 of *Springer-Verlag LNCS*, pages 191–202, 2003.
13. D. Meredith, G.A. Wiggins, and K. Lemström. Pattern induction and matching in polyphonic music and other multi-dimensional datasets. In *the 5th World Multi-Conference on Systemics, Cybernetics and Informatics (SCI'2001)*, volume X, pages 61–66, Orlando, FLO, July 2001.
14. E. Ukkonen, K. Lemström, and V. Mäkinen. Sweepline the music! In *Computer Science in Perspective*, volume 2598 of *Springer-Verlag LNCS*, pages 330–342, 2003.
15. E. Ukkonen and D. Wood. Fast approximate string matching with suffix automata. *Algorithmica*, 10:353–364, 1993.