Computing and Informatics, Vol. 28, 2009, 729–744

A HYBRID ALGORITHM FOR THE LONGEST COMMON TRANSPOSITION-INVARIANT SUBSEQUENCE PROBLEM

Sebastian Deorowicz

Institute of Informatics
Silesian University of Technology
Akademicka 16, 44-100 Gliwice, Poland
e-mail: sebastian.deorowicz@polsl.pl

Szymon Grabowski

Computer Engineering Department
Technical University of Łódź
al. Politechniki 11, 90-924 Łódź, Poland
e-mail: sgrabow@kis.p.lodz.pl

Manuscript received 16 July 2008; revised 23 June 2009
Communicated by Marián Vajteršič

Abstract. The longest common transposition-invariant subsequence (LCTS) problem is a music information retrieval oriented variation of the classic LCS problem. There are basically only two known efficient approaches to calculate the length of the LCTS, one based on sparse dynamic programming and the other on bit-parallelism. In this work, we propose a hybrid algorithm picking the better of the two algorithms for individual subproblems. Experiments on music (MIDI), with 32-bit and 64-bit implementations, show that the proposed algorithm outperforms the faster of the two component algorithms by a factor of 1.4–2.0, depending on sequence lengths. Similar, if not better, improvements can be observed for random data with Gaussian distribution. Also for uniformly random data, the hybrid algorithm is the winner if the alphabet is neither too small (at least 32 symbols) nor too large (up to 128 symbols). Part of the success of our scheme is attributed to a quite robust component selection heuristic.

Keywords: Longest common transposition-invariant subsequence (LCTS), bit-parallelism, sparse dynamic programming, string matching

Mathematics Subject Classification 2000: 68W05

1 INTRODUCTION

One of the classic problems in the field of string matching concerns the longest common subsequence (LCS) of two sequences. The problem can be stated like this: Given two sequences: $A = a_0, \ldots, a_{m-1}$ and $B = b_0, \ldots, b_{n-1}$, over an alphabet $\Sigma = \{0, \ldots, \sigma-1\}$, report the length ℓ of a(ny) longest subsequence $\langle a_{i_1}, a_{i_2}, \ldots, a_{i_\ell}\rangle$ of A, where $i_k < i_{k+1}$ for all $k < \ell$, $a_{i_1} = b_{j_1}, a_{i_2} = b_{j_2}, \ldots, a_{i_\ell} = b_{j_\ell}$, and $j_k < j_{k+1}$ for all $k < \ell$. In other words, the output string is obtained from removing zero or more characters from both A and B, i.e., is a subsequence of both A and B, and no longer string with this property exists. W.l.o.g. we assume $m \leq n$. Moreover, we assume that $\sigma = O(n)$. An extended version of the problem also asks for the sequence itself (which does not have to be unique), not just its length. Note that the LCS problem is a dual of the indel distance, capturing the number of insertions and deletions needed to transform one sequence into another: $2\,\mathrm{LCS}(A, B) = n + m - id(A, B)$ [14]. It is however accepted to speak about LCS when we are interested in finding the global measure of similarity, and the indel distance (being a dissimilarity measure) when a pattern shifts over a (much) longer text. Frequent applications of the LCS measure include DNA sequence analysis and comparisons of different versions of program source files (Unix *diff* tool). The classic dynamic programming (DP) algorithm for LCS has $O(mn)$ time complexity, and, surprisingly, not much better complexities are known for this problem for the worst case. LCS has been thoroughly explored [5]. Also, numerous variations of the problem have been posed, see e.g. [14] for details.

The variation of LCS which is the concern of the current work was introduced in 2000 [17] and has applications in music information retrieval. An important trait of similar melodic sequences is that they can differ in the key, but humans perceive them as same melodies. More formally, the problem of *longest common transposition-invariant subsequence* (LCTS) is to find the length of a(ny) longest subsequence $\langle a_{i_1}, a_{i_2}, \ldots, a_{i_\ell}\rangle$ of A such that $a_{i_1} = b_{j_1}+t, a_{i_2} = b_{j_2}+t, \ldots a_{i_\ell} = b_{j_\ell}+t$, for some $-\sigma < t < \sigma$. In other words, we look for the length of the longest subsequence of A and B matching according to any *transposition*. The alphabet size in music (MIDI) application is usually 128. A naïve algorithm for calculating LCTS is to run the dynamic programming algorithm independently for each transposition, which yields $O(mn\sigma)$ time. Almost all known better solutions belong to one of the two categories: they are bit-parallel or based on sparse dynamic programming.

Bit-parallelism [4] is a widely used technique (in many string matching problems) making use of the simple fact that any real CPU works on many bits in parallel (usually 32 or 64 nowadays). For LCS, there are known algorithms of $O(n\lceil m/w\rceil)$

worst case time complexity [1, 7, 16], where w is the machine word size (in bits). Adopting any of those algorithms for the LCTS problem is straightforward: it is enough to run the LCS routine for each of the $2\sigma - 1$ transpositions separately, achieving $O(n\sigma\lceil m/w\rceil)$ time complexity, not counting a preprocessing stage (to be discussed later). Experiments show [8] that this approach is quite practical.

Sparse dynamic programming (SPD) [18] is a technique of visiting only a subset of the DP matrix, namely those cells that correspond to matching pairs of characters of A and B. For the LCS problem, this technique was first used in the seminal paper by Hunt and Szymanski [15], and to apply it for the LCTS it was basically enough to notice that each cell in the DP matrix corresponds to exactly one transposition. This technique, subjected to a couple of refinements, allowed to obtain $O(mn \log m)$, $O(mn \log\log m)$ [18], $O(mn \log \sigma)$ [13], and finally $O(mn \log\log \sigma)$ [21, 8] time complexity. Also, in [8] Deorowicz presented a related variant, slightly worse in theory, with $O(mn\lceil \log \sigma / \log w\rceil)$ worst-case time, which however won in his thorough tests for MIDI and for uniformly random data over a large enough alphabet (say, of size 96 or more). In this paper, we present a hybrid algorithm for LCTS making use of a simple observation: if the alphabet is small, the bit-parallel approach is a clear winner, but for large enough alphabets sparse dynamic programming algorithms start to dominate. Our idea is to use the bit-parallel technique for frequent transpositions and the Hunt–Szymanski algorithm for the rare ones. We assume the AC^0 RAM model of computation [3], in which the machine word has at least $\log n$ bits (for any considered sequence lengths, n and m, $n \geq m$) and the allowed constant-time operations contain standard arithmetics (without multiplications and divisions) and bitwise operations (and, or, shifts, etc.). Experiments in Section 7 on MIDI and random data confirm attractiveness of this simple approach.

A preliminary version of this paper was presented in [12].

2 THE HUNT-SZYMANSKI ALGORITHM FOR LCS

A simple idea introduced in 1977 by Hunt and Szymanski [15] has set the ground for the theoretically best LCS algorithms [2, 10], and also for best LCTS algorithms based on sparse dynamic programming. In this section we present the HS algorithm in detail.

We start with a definition. We say that a cell (i, j) of the dynamic programming matrix M stores a match of rank k iff $a_i = b_j$ and $\mathrm{LLCS}(a_{1..i} = b_{1..j}) = k$. Now we can present the algorithm.

Let the matrix M have $m + 1$ rows and $n + 1$ columns. In the preprocessing we create lists of successive occurrences of all alphabet symbols in the shorter sequence, A. This requires $O(m+\sigma)$ space and time, which is bounded by $O(n)$ in our case. Note that after this stage for each character of B we can access the occurrence list of this character in A in constant time. Traversing a list obviously requires $O(1)$ time per item. For a technical reason, we will scan the lists in the reverse order, i.e., corresponding to right-to-left scans (with skips) over the rows of M.

We also maintain an array T, which stores at position t the leftmost seen-so-far column with a match of rank t.

At the beginning this array is zeroed. Throughout the whole processing we store the index of the last non-zero cell in T in a variable $k_{\max}$.

Now, we visit the matching cells of M, rowwise and from right to left in rows, using the lists obtained in the preprocessing stage. Let a considered match be at cell (i, j), where i denotes the row and j denotes the column. We look for the minimum index t such that $T[t] \geq j$. If there is no such index t, that is, $T[h] < j$, for $h = 1..k_{\max}$, then we set $T[k_{\max} + 1] \leftarrow j$ and increment $k_{\max}$ by one. In the opposite case we distinguish between $T[t] = j$ and $T[t] > j$. The equality means that the current match has the same rank as some match at the same column but in an earlier row, i.e., the current match does not imply any update to T. If however $T[t] > j$, then we set $T[t] \leftarrow j$, as there must be that $T[t-1] < j$ (if only $t > 1$) and there hasn't been yet a match with rank t in the jth (or earlier) column. Note that because of the right-to-left scan order, there can be several updates to a single cell of T within a single row of M. The desired LLCS is the value of $k_{\max}$ after finishing the last row.

Let r denote the number of all matches in M. It is easy to notice that the time complexity of the algorithm depends on how fast one can find, for each of r matches, the proper t to satisfy the aforementioned inequality. The plain binary search immediately leads to $O(n + r \log m)$ time (the additive term n is from visiting all the matrix rows, even if empty), but since the non-empty range of T never has more than $\ell = \mathrm{LLCS}(A, B)$ elements, it is more precise to express the worst case complexity as $O(n + r \log \ell)$. Note that we can ignore the preprocessing cost since it is never dominating. Note also that $r = O(mn)$ in the worst case.

3 THE DEOROWICZ VARIANT

The Hunt-Szymanski concept was inspiration for a number of subsequent algorithms for LCS calculation. Finding the rank of a match can be performed in a more refined way than with binary search, in particular, using the van Emde Boas (vEB) dynamic data structure [9] which is applicable if the universe of keys is nicely bounded; a possibility noticed already by Hunt and Szymanski in their original work. In our problem, this translates to $O(n + r \log \log m)$ worst case complexity. There are even better (and more complex) theoretical solutions [2, 10] from the Hunt–Szymanski family, where for example the symbol r is replaced with D, the number of so-called dominant matches ($D \leq r$).

In this section, we outline a practical HS variation by Deorowicz [8], which was used in the cited work for calculating LCTS in $O(mn\lceil \log \sigma / \log w \rceil)$ worst-case time (in an algorithm denoted there as OUR-3).

The HS routine is based on finding the successor of the current column index in the array T. The idea from [8] was to support the successor queries with a w-ary tree, where w is the machine word size (in bits). More precisely, the w-ary tree

is a complete tree of arity w, storing unique keys from the range $0, \ldots, v-1$, in which each node is an array of exactly w bits. In the RAM model of computation, $w = \Theta(\log n)$, where n is, roughly speaking, the length of the longest addressable text. Because of its regularity, the w-ary tree can be implemented without any pointers (note also that the keys do not hold any satellite information). The height h of this tree is $O(\lceil \log v / \log w \rceil)$.

In Deorowicz's LCTS algorithm, w-ary trees are used to store the values of the T arrays for individual transpositions. In total, $2\sigma - 1$ w-ary trees are maintained. To check if j is in the tree (for a given transposition), it is enough to examine one particular bit in a certain leaf, which takes $O(1)$ time. Inserting or removing a value needs to set or reset the corresponding bit in the leaf and update the nodes upward the tree, with the overall complexity of $O(h) = O(\lceil \log v / \log w \rceil)$. The successor operation for j requires looking for the next set bit in the leaf corresponding to value j (which can be done in constant time), and if there is no such set bit, moving upward the tree and following analogously until such a bit is found (or it is found that there is no value greater than j in the tree). Because each node is handled in $O(1)$ time, the overall time complexity is again $O(\lceil \log v / \log w \rceil)$. Figure 1 illustrates. A straightforward solution would take $v = m$, but in the cited work it was shown how to decrease v to $\min(\sigma, m)$, which is beneficial both for speed and storage occupancy.

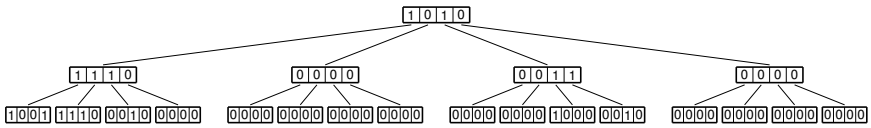

Fig. 1. The w-ary tree [8], $w = 4$

3.1 The Case of Large w

In the presented variant it was assumed that $w = \Theta(\log n)$ and then finding the successor (the next set bit) within a machine word can be achieved with aid of a lookup table of, e.g., $O(\sqrt{n})$ size, in constant time. Still, the case of a large w (a scenario which is getting realistic in recent years) requires further considerations.

We take a look at two important cases. In one we have $w = O(\log^{O(1)} n)$, where the exponent is greater than 1. We can then work with only a prefix (or any factor for that matter) of such a word of length $\log n$ bits, as $\log w$ remains $O(\log \log n)$, so the time complexity is as above. The other interesting case is $w = O(n^{\varepsilon})$. What we can do here is to use the algorithm by Brodnik et al. [6] which identifies the least significant set bit of a word in $\Theta(\log \log w)$. Interestingly, they prove that their time complexity bound is tight, if e.g. multiplications and divisions are forbidden (we note that some modern CPUs contain a constant-time opcode for the number

of leading zeros, which can be a practical solution, but goes beyond the assumed AC^0 RAM model). Adopting this problem to our setting is straightforward, using bit masking. As a result, we need to spend $\Theta(\log\log w)$ time in a node, which leads to $O(\lceil \log\sigma \log\log w/\log w\rceil)$ overall time per matrix cell, for any $w = \omega(\log^{O(1)} n)$. Assuming $w = O(n^{\varepsilon})$, we obtain $O(\lceil \log\sigma \log\log n/\log n\rceil)$ time. We note in passing that erasing the next set bit (or e.g. the least significant bit in a word) can be easily performed in $O(1)$ time using a folklore trick but in a w-ary tree we could use this idea only in leaves; in non-leaves we must know the bit position too, as it points to the respective child of the current node.

4 BIT-PARALLEL APPROACH

Another approach to the LCS (and subsequently to LCTS) problem is based on the well-known property of the dynamic programming matrix M: two adjacent values in a row, or in a column, differ by at most 1. Thanks to this property, efficient bit-parallel LCS-solving algorithms can be devised, and the first of them was presented by Allison and Dix [1]. If the length of the shorter of the two sequences is not greater than the machine word size (in bits), then the algorithm runs in linear time (not counting the preprocessing). This is not always the case, of course, but longer bit-vectors, representing one of the sequences can be simulated using several machine words. In general, the time complexity of this algorithm is $O(n\lceil m/w\rceil)$ and does not depend on the content of the two sequences.

Future attempts along these lines, by Crochemore et al. [7] and Hyyrö [16], were to simplify and speed-up the bit-parallel computation formulae, but the algorithm complexity remained.

All those variants are based on preprocessing using $O(\sigma\lceil m/w\rceil + m)$ time and $O(\sigma m)$ bits of space. In that phase, σ bit vectors PM_λ of size m are generated, where for any alphabet symbol λ, the bit $\mathrm{PM}_\lambda[i]$ is set iff $A_i = \lambda$.

In the main loop of the Allison–Dix algorithm, there are six operations (here and later: assignment operations not counted) per a character of B. This was reduced to five operations in the Crochemore et al. algorithm [7], which was successively reduced by Hyyrö [16] to four operations, without e.g. table lookups except for the references to vectors PM_λ.

In Hyyrö's experiments (AMD Athlon64 with $w = 64$), the Allison–Dix algorithm was the slowest among the three bit-parallel variants, but the Crochemore et al. algorithm was faster by only 5 %, while the algorithm from Figure 2 was faster than the Crochemore et al. by 15 %.

5 FROM LCS TO LCTS

Mäkinen et al. [18] made a simple observation: each cell in M corresponds to exactly one transposition in the LCTS problem. This means that the technique of Hunt-Szymanski (in virtually any possible variation) can be separately applied for each

```
{Preprocessing}
01  for λ ∈ Σ do PM_λ ← 0^m
02  for i ← 0 to m − 1 do PM_{a_i} ← PM_{a_i} | 0^{m−i−1}10^i
{Computing LCS}
03  V ← 1^m
04  for j ← 0 to n − 1 do
05      U ← V & PM_{b_j}
06      V ← (V + U) | (V − U)
{Calculating number of 0's in V}
07  r ← 0;   V ← ~V
08  while V ≠ 0^m do
09      r ← r + 1;   V ← V& (V − 1)
10  return r
```

Fig. 2. Hyyrö's bit-parallel algorithm [16] for LCS(A, B) computation. The preprocessing is performed in lines 01–02. The meaning of the binary operators `&` , `|` , `~` is like in the C programming language.

of $2\sigma - 1$ transpositions. The total amount of matches is exactly mn, and this easily implies the time complexity of $O(mn \log \ell)$, or $O(mn \log \log m)$ in a more theoretical version (we neglect the preprocessing here, which is also not problematic under typical assumptions). More recent results, including the practical Deorowicz's algorithm described in Section 3, have been listed in Section 1; they all are based on sparse dynamic programming.

It is even simpler to switch from LCS to LCTS using the bit-parallel algorithms: the procedure is run for each transposition separately, yielding the extra σ multiplicative factor. Albeit this can be called a brute-force technique, it fares surprisingly well for the MIDI domain. What needs a short discussion is the preprocessing for bit-parallel LCTS algorithms. In a naïve variant, the LCS preprocessing routine is simply run for each of the $2\sigma - 1$ transpositions, yielding overall $O(\sigma^2 \lceil m/w \rceil + m\sigma)$ time. This, however, can be easily improved to $O(\sigma \lceil m/w \rceil + m)$ time (and similarly the space can be reduced), by merely using LCS preprocessing for the LCTS problem. The trick is to modify a bit the search phrase; for a given transposition t and the current column corresponding to symbol b_j, the current bit-vector PM_λ is taken for the alphabet symbol $\lambda = b_j - t$ (provided that the resulting symbol is within the alphabet range). This idea was found practical and is used in our implementation, to speed-up the calculations in the BP component by up to 2 %. Note that the number of non-zero PM_λ vectors is at most $\min(m, \sigma)$, and for the remaining alphabets symbols (if any) there is no need to even allocate space for the bit-vectors; the price we pay for it is an additive $O(m + \sigma)$ term, which leads to overall preprocessing time $O(\min(m, \sigma)\lceil m/w \rceil + m + \sigma)$.

Less trivially, the preprocessing can be improved even more, to $O(m)$ worst-case time, for the price of increasing the space by a constant factor. This can be achieved from getting rid of zeroing the bit vectors even once. To this end, we use

the old array initialization trick [19, Section III 8.1], which however requires extra arrays, increasing the space up to three times. Recently, Navarro [20] presented a space-efficient variant of this idea, with only $n + o(n)$ extra bits needed for an array with n arbitrary items, on a RAM machine, which translates in our problem to $\lceil m/w \rceil + o(\lceil m/w \rceil)$ bits.

We haven't implemented this idea (in any variant) but we don't anticipate it to be practical, as any access to PM_λ must be translated to three array accesses (in the original, more space-demanding variant), if the trick in question is applied.

6 OUR ALGORITHM

It is easy to notice that the two presented approaches significantly differ in their characteristics: the algorithms from the Hunt–Szymanski family are efficient when matches in the dynamic programming table are infrequent, while bit-parallel algorithms are insensitive to the distribution of the input data. When we focus on LCTS rather than LCS, however, it is wiser to say that those two approaches are not simply different: they can be *complementary*. The bit-parallel (BP) approach for LCS adapted for the LCTS problem runs in time directly proportional to the alphabet size, but its running time for each alphabet symbol (i.e., transposition in that case) is approximately the same. This is not the case with HS, where processing infrequent transpositions is faster than the frequent ones.

Here comes our simple idea: use HS for transpositions with small enough number of occurrences, and the bit-parallel approach for the remaining ones. Now, we have to find a relevant threshold to properly distinguish between "HS-friendly" and "BP-friendly" transpositions.

We start with counting the number of cells corresponding to each transposition, which can be done in $O(n+m+\sigma^2)$ time, and we sort the transpositions according to their frequency, in $O(\sigma \log \sigma)$ or $O(\sigma \log_\sigma n)$ time, the latter with use of radix sort. Let us note in passing that the straightforward $O(n + m + \sigma^2)$-time algorithm for gathering match counts for transpositions can be improved to $O(n+m+\sigma \log \sigma)$, with FFT. To this end, we create two arrays, C_A and C_B, of $2\sigma-1$ cells, one for sequence A and one for B, which store the number of occurrences of alphabet symbols in A and B, respectively, in their first halves, and are padded with zeros elsewhere. Now, the cyclic correlation of discrete sequences C_A and C_B can be calculated in $O(\sigma \log \sigma)$ time, using FFT, and its $2\sigma-1$ coefficients are the numbers of occurrences of all $2\sigma-1$ transpositions, i.e., exactly what we needed. Note that this $O(\sigma \log \sigma)$ complexity assumes constant-time multiplications, which is fulfilled "in practice", but not in the AC^0 RAM model (cf. Section 1), where this should be multiplied by $\log \log n$ yet. Fortunately, we can achieve $O(1)$-time multiplications in our case, using a lookup table with precomputed products for all pairs of $\lceil (\log_2 n)/3 \rceil$-bit integers, which is enough to obtain the product of numbers having 3 times more bits in constant time. Building the lookup table takes $o(n)$ time, i.e., can be neglected in overall time complexity.

In the following, we need some extra notation. Let $R(t)$, $0 \leq R(t) < 2\sigma - 1$, be the rank of a transposition t, that is, its position in the sorted order; $R(t) = 0$ if t is the most frequent (or any of them if more than one) transposition. Also, let $f(t)$ denote the number of occurrences of transposition t. Now we perform the preprocessing routines for the BP and HS algorithms, as described in previous sections, and after that we are ready to run the BP and HS routines for some sampled transpositions. Namely, we take the transposition t_1 of rank 0, and transposition t_2 of rank $\lceil (2\sigma-1)/3 \rceil - 1$ and run the BP algorithm for them, measuring the execution times, $time(t_1)$ and $time(t_2)$. We remove t_1 and t_2 from the transposition list, and (conceptually) refresh the rank list. Similarly, we take the transposition t_3 of rank $\lceil (2\sigma - 3)/3 \rceil - 1$ and t_4 of rank $2\sigma - 4$ (i.e., the least frequent), and run the HS algorithm for them, storing the execution times. We assume that for both component algorithms, BP and HS, their execution time is linear in the number of matches, for fixed values of n, m, and σ (this is only an approximation, but according to our preliminary experiments, it works fairly well in practice). Of course, the dependence on the number of matches is clear for the HS algorithm, while it is less obvious (and much weaker) for the BP algorithm, but still it exists, due to some machine-dependent issues (the more matches, the slower the BP algorithm works). The four time measurements are enough to pass two straight lines whose intersection point will determine the range of applicabilities for both components. Namely, we pass a straight line through the points $(f(t_1), time(t_1))$ and $(f(t_2), time(t_2))$, and another straight line through $(f(t_3), time(t_3))$ and $(f(t_4), time(t_4))$. Transpositions with the number of matches at least as large as the first coordinate of the intersection point will be handled by the BP algorithm, and those with fewer matches by the HS algorithm. Naturally, we do not handle those four sampled transpositions again. An extra improvement comes from the idea of premature stoppage in the HS routine; namely, if the algorithm for a given transposition t cannot beat the result of the best-so-far transposition (even in the most optimistic case when each of the remaining rows increases LLCS(t) by 1), it is stopped. (Although not mentioned there, that idea was also used in [8].) Statistically, denser transpositions have greater chance to yield long common subsequences, therefore we first use the BP component, then HS, starting from denser transpositions; as a result the effectiveness of the stoppage idea grows.

Finally, we note that the overall time of the hybrid algorithm is $O(\sigma n \lceil m/w \rceil + mn \lceil \log \sigma / \log w \rceil)$, including the preprocessing costs of both components, in the worst case (assuming $w = O(\log^{O(1)} n)$; the case of larger w was discussed in Subsection 3.1).

7 EXPERIMENTAL RESULTS

We have run several experiments to evaluate the performance of our algorithm against its strongest competitors. The experiments were carried out on an AMD Athlon64 5 000+ (CPU clock 2 600 MHz) machine with 2 GB of RAM, running Win-

dows Vista 64 operating system. We have implemented all the algorithms in C++, and compiled with Microsoft Visual C++ 2005. As we apply bit-parallelism, the performance between the case of $w = 32$ and $w = 64$ should differ significantly, which we confirm, presenting the results of 32- and 64-bit implementations.

We considered three cases: running the algorithms (*i*) on music data, (*ii*) on uniformly random data, and (*iii*) on random data with Gaussian distribution; in the latter two cases, for varying alphabet size. The mean of the Gaussian distribution was always set in the middle of the alphabet and the standard deviations to $\sigma/8$.

For the first set of experiments we used a concatenation of 7 543 music pieces, obtained by extracting the pitch values from MIDI files. The total length is 1 828 089 bytes. The pitch values are in the range $0, \ldots, 127$, which corresponds to 255 possible transpositions. This data is far from random: the six most frequent pitch values occur 915 082 times, which is approximately 50 % of the whole text, and the total number of different pitch values is just 55. Consequently, the number of possible existing "transpositions", i.e., differences between any pairs of characters from two different excerpts of this file, is much lower than the theoretical maximum of 255. This dataset was previously used in the literature (e.g., [8, 11]), for various MIR-oriented problems, including LCTS.

A set of 101 pairs of randomly extracted excerpts from the text was generated. We varied the lengths, n and m, of those sequences, but always set $n = m$. The reported times (and other measurements, like chosen thresholds) are the medians over all 101 trials. Figure 3 demonstrates the relation between the (percentage) amount of most frequent transpositions and the amount of matches covered by them. We can see, for example, for the MIDI data, that the top 20 % of the existent transpositions (sorted by frequency) already cover more than half of the matches while 60 % of the existent transpositions are enough to cover over 90 % of matches. For the uniformly random data, as expected, the curve is more flat.

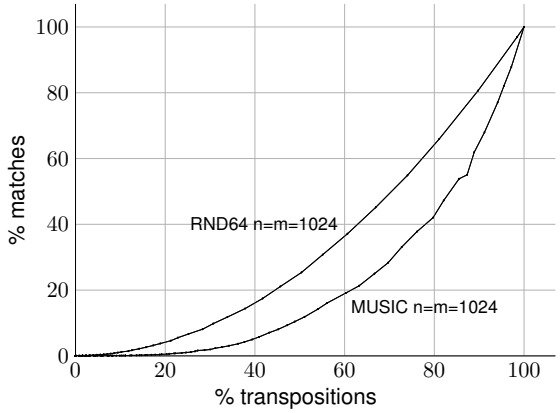

Fig. 3. % matches vs. % transpositions

Figure 4 shows the overall processing time of our hybrid algorithm as a function of the minimal number of matches in transpositions handled by the HS component. Basically the same phenomenon, as a function of varying alphabet size (only for the two random distributions), is also presented in Figure 5. Note that extreme parameters of the thresholds trigger a single component for all transpositions; in most cases, for alphabet size up to 64 or 128 (depending on the dataset and whether the implementation is 32- or 64-bit), the "single best" component is the BP algorithm, while for larger alphabets the HS algorithm starts to win. It can be also seen that the Gaussian data are much more sensitive to small changes of the thresholds (in their low ranges), which is not surprising.

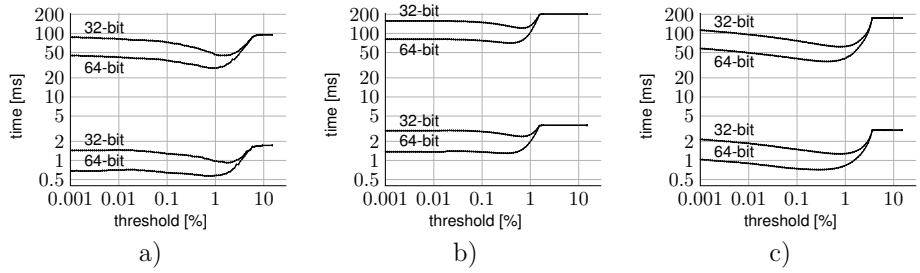

Fig. 4. Overall processing time of the hybrid with varying threshold of the minimal number of matches in transpositions handled by the HS component. a) MUSIC, b) RANDOM-64, c) GAUSS-64. Top pairs of curves for $n = m = 4\,096$, bottom pairs for $n = m = 512$

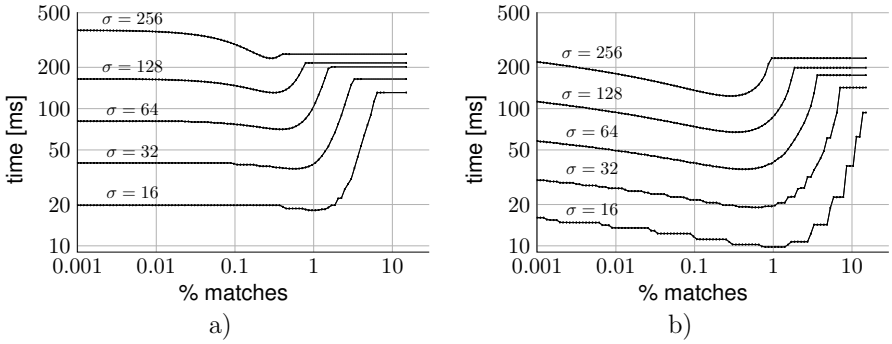

Fig. 5. Overall processing time of the hybrid with varying threshold of the total percentage of matches handled by the HS component (transpositions ordered from the sparsest to the densest). $n = m = 4\,096$, $\sigma = 16, \ldots, 256$, 64-bit implementation: a) RANDOM, b) GAUSS

It occurs that the best split for the music data allots about 80–90 % matches (from the most frequent transpositions) to the BP algorithm, while the remaining

10–20 % matches are handled by the HS variant (cf. Table 1–4). In other words (cf. Figure 3), less than 40 % of the most frequent transpositions should be processed by BP. Note also that the BP component is faster by about 25 % (i.e., needs about 20 % less time) than the HS component (if both are applied exclusively) for music data and $w = 32$. For the 64-bit implementation the advantage of the BP component even grows to 2-fold. For the uniformly random data ($\sigma = 64$), those differences are even greater, but drop rapidly with growing alphabet size (on the other hand, for σ smaller than 64, the advantage of the BP algorithm is even more striking).

We also evaluated an "oracular" component selection (column *Best time* in all tables), which sets the lower bound for any selecting heuristic. It can be seen that our idea based on crossing lines is quite stable across all experiments and the loss to the lower bound is usually within 2 % of overall time (the worst case is in Table 2, $n = m = 256$, where our time was by over 6 % longer than with using the oracle).

$n = m$	BP time [ms]	HS time [ms]	Best time [ms]	Hyb. time [ms]	Best thr. [%]	Hyb. thr. [%]	BP trans [%]	BP match [%]
256	0.3157	0.5167	0.2686	0.2768	1.6928	1.3930	45.4	83.8
512	1.4366	1.7222	0.9210	0.9686	1.8621	1.5752	42.1	80.5
1 024	5.5727	6.3598	3.0630	3.1441	1.6928	1.4759	40.2	81.5
2 048	23.2568	25.5058	13.3131	13.5623	1.5389	1.4969	34.8	78.3
4 096	91.7169	95.0223	44.9423	45.6610	1.3990	1.4930	34.7	80.9
8 192	381.6627	381.3309	185.5993	188.4778	1.5389	1.5642	33.6	79.9

Table 1. MUSIC, 32-bit implementation

$n = m$	BP time [ms]	HS time [ms]	Best time [ms]	Hyb. time [ms]	Best thr. [%]	Hyb. thr. [%]	BP trans [%]	BP match [%]
256	0.1842	0.5183	0.1842	0.2151	0.0000	0.7371	61.9	92.4
512	0.6811	1.7288	0.5669	0.5868	0.8687	0.7022	61.2	93.1
1024	3.0936	6.3303	2.0560	2.1032	1.1562	0.8202	55.0	91.1
2048	12.2291	25.5285	8.3442	8.6279	0.8687	0.7895	52.1	90.1
4096	47.1900	95.0745	28.4840	29.0666	0.7897	0.7683	48.9	91.8
8192	193.8347	380.7648	112.5319	113.1798	0.8687	0.7952	43.7	91.2

Table 2. MUSIC, 64-bit implementation

$n = m$	BP time [ms]	HS time [ms]	Best time [ms]	Hyb. time [ms]	Best thr. [%]	Hyb. thr. [%]	BP trans [%]	BP match [%]
256	0.7996	1.4635	0.7677	0.7771	0.1890	0.1898	75.6	93.6
512	2.7441	4.5429	2.4480	2.4640	0.2287	0.2248	71.3	91.2
1024	11.8452	15.4458	9.6989	9.7430	0.3044	0.2938	62.3	85.3
2048	42.9331	56.6781	34.5518	34.7321	0.3044	0.2914	62.3	85.6
4096	164.4345	215.2981	130.7266	131.3415	0.3044	0.2933	62.3	85.3
8192	729.7012	877.4197	572.9071	576.5836	0.3349	0.3295	57.6	81.8

Table 3. RANDOM-128, 64-bit implementation

The detailed timings of the algorithms: HS, BP and our hybrid, are given in the tables, for 32-bit and 64-bit implementations separately. In rows, the problem size changes from $n = m = 256$ to $n = m = 8192$. The right half of the columns requires a brief explanation. *Best threshold* specifies the minimum fraction of matches per

$n = m$	BP time [ms]	HS time [ms]	Best time [ms]	Hyb. time [ms]	Best thr. [%]	Hyb. thr. [%]	BP trans [%]	BP match [%]
256	0.5463	1.1105	0.4193	0.4316	0.1420	0.2572	51.1	94.3
512	1.9991	3.7308	1.3030	1.3254	0.1890	0.2741	46.7	94.0
1024	9.2762	13.3282	5.0989	5.1376	0.3349	0.3455	40.3	92.3
2048	35.9229	50.6291	18.0331	18.1210	0.3349	0.3336	38.0	92.6
4096	144.2099	198.6377	67.3530	67.4932	0.3044	0.3349	36.1	92.7
8192	597.2110	778.5215	263.3859	263.5459	0.3684	0.3506	34.3	92.3

Table 4. GAUSS-128, 64-bit implementation

transposition, for which the BP algorithm starts to work faster than the HS algorithm (if used for this transposition). *Hybrid threshold* conveys similar information; the difference is that here we see the threshold selected by our component selection heuristic. Roughly, the closer those two threshold values are, for a given problem instance, the better the heuristic is expected to work. [1]

The next column, *BP transpositions* is the fraction of transpositions handled by the BP component, using our selection heuristic. Finally, the column *BP match* holds the fractions of matches in the transpositions for which the BP algorithm is triggered.

The speedup factor of the hybrid algorithm over the better of the two components (i.e., BP) on the music data varies from 1.36 ($n = 256$) to 1.97 ($n = 8192$) in the 32-bit implementations, and from 1.11 ($n = 256$) to 1.71 ($n = 8192$) in the 64-bit implementations, so it improves with growing n. Note that we skip non-existent transpositions in the BP algorithm, which boosts its performance on the music data very significantly.

On uniformly random data, the situation is somewhat different. For small to moderate σ (up to 64) the bit-parallel algorithm is much faster than the HS one (note the scale on Figure 5), even in the 32-bit version the difference is 4-fold in case of $\sigma = 16$ and $n = 4\,096$ (switching to 64 bits makes is almost 7-fold), but the picture changes for $\sigma = 128$ and $\sigma = 256$. Interestingly, for small alphabets the HS component beats the BP component on some (few) transpositions, so the hybrid, with the threshold selected properly, again appears better than both its components (but with the speedup of about 10 % only, at best). For a large enough alphabet ($\sigma = 256$) the BP algorithm usually can win on no transposition, hence the "optimal" hybrid degenerates into the HS component. The border case is $\sigma = 128$ where HS takes the lead but its advantage over BP is quite moderate; in that case

[1] The presented thresholds are medians from individual experiments, but there is a subtle difference between "best times" and "hybrid times". The column *Best time* uses a single threshold (the one for which the median time over 101 runs is minimized), while for the column *Hybrid time* individual thresholds for each test run are used (and the median times for runs with those possibly different thresholds are presented). Although insignificant in practice, this could, in theory, lead to surprising effects, e.g., a shorter *Hybrid time* than *Best time*. The reason for which we chose this presentation methodology was to make it compatible with Figures 4 and 5, where a single "best threshold" had to be used.

the hybrid algorithm is faster than HS by 12–24 % (easy to guess, the speedups close to 24 % are for the 32-bit implementations).

The HS algorithm does not change its speed when switched from 32 to 64 bits, while the improvement is obvious for the BP algorithm, and the speedup factor varies from about 1.6 to 1.9. The stages of the algorithm which do not gain from longer registers are the preprocessing and the final counting of the set bits in vector V (cf. Figure 2). The columns *BP transpositions* and *BP match* confirm that after switching from the 32- to 64-bit implementation, the bit-parallel component is selected more often.

8 CONCLUSIONS

We presented a simple hybrid algorithm for the longest common transposition-invariant problem, choosing "the best of the two worlds": bit-parallel and sparse dynamic programming approaches. Experiments confirm practicality of this idea, especially on real music (MIDI) data, where the LCTS problem has a natural application. Our hybrid requires a (fast and reliable) criterion for selecting the component for each transposition. We solved this problem giving a simple heuristic, which typically loses not more than 2 % time compared to the selection with an oracle.

On the overall, the proposed algorithm outperforms the faster of the two component methods on the MIDI data by a factor of 1.4–2.0 in 32-bit implementations and 1.1–1.7 in 64-bit implementations, where the larger gaps are for longer sequences. On the uniformly random data the improvements are smaller, and they rarely exceed the factor 1.2. The gains are greater for Gaussian distribution of data, and they sometimes exceed the factor 2.0, even in the 64-bit implementations.

Acknowledgements

The research of this project was partially supported by the Minister of Science and Higher Education grant 3177/B/T02/2008/35 (first author) and a habilitation grant (2008–2009) of Rector of Technical University of Łódź (second author).

REFERENCES

[1] Allison L.—Dix T. L.: A Bit-String Longest Common Subsequence Algorithm. Information Processing Letters, Vol. 23, 1986, No. 6, pp. 305–310.

[2] Apostolico A.—Guerra C.: The Longest Common Subsequence Problem Revisited. Algorithmica 2, pp. 316–336, 1987.

[3] Allender E.: Circuit Complexity before the Dawn of the New Millennium. Proc. of the 16th Conf. on Foundations of Software Technology and Theoretical Computer Science, 1996, pp. 1–18.

[4] Baeza-Yates, R. A.: Efficient Text Searching. Ph. D. thesis. Department of Computer Science, University of Waterloo, May 1989. Also as Research Report CS-89-17.

[5] Bergroth, L.—Hakonen, H.—Raita, T.: A Survey of Longest Common Subsequence Algorithms. Proc. of the 7th Int. Symp. on String Processing and Information Retrieval (SPIRE), 2000, pp. 39–48.

[6] Brodnik, A.—Miltersen, P. B.—Munro, J. I.: Trans-Dichotomous Algorithms Without Multiplication – Some Upper and Lower Bounds. Proc. of the 5th Int. Workshop on Algorithms and Data Structures (WADS), 1997, pp. 426–439.

[7] Crochemore, M.—Iliopoulos, C. S.—Pinzon, Y. J.—Reid, J. F.: A Fast and Practical Bit-Vector Algorithm for the Longest Common Subsequence Problem. Information Processing Letters, Vol. 80, 2001, No. 6, pp. 279–285.

[8] Deorowicz, S.: Speeding up Transposition-Invariant String Matching. Information Processing Letters, Vol. 100, 2006, No. 1, pp. 14–20.

[9] van Emde Boas, P.—Kaas, R.—Zijlstra, E.: Preserving Order in a Forest in Less Than Logarithmic Time and Linear Space. Information Processing Letters, Vol. 6, 1977, No. 3, pp. 80–82.

[10] Eppstein, D.—Galil, Z.—Giancarlo, R.—Italiano, G. F.: Sparse Dynamic Programming I: Linear Cost Functions. J. of the ACM, Vol. 39, 1992, No. 3, pp. 519–545.

[11] Fredriksson, K.—Mäkinen, V.—Navarro, G.: Flexible Music Retrieval in Sublinear Time. International Journal of Foundations of Computer Science Vol. 17, 2006, No. 6, pp. 1345–1364.

[12] Grabowski, Sz.—Deorowicz, S.: Nice To Be A Chimera: A Hybrid Algorithm For The Longest Common Transposition-Invariant Subsequence Problem. In Proceedings of the 9th International Conference on Modern Problems of Radio Engineering, Telecommunications and Computer Science (TCSET 2008), Lviv-Slavsko, Ukraine, 2008, pp. 50–54.

[13] Grabowski, Sz.—Navarro, G.: $O(mn \log \sigma)$ Time Transposition Invariant LCS Computation. Technical Report TR/DCC-2004-6, University of Chile, Department of Computer Science, September 2004, `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/transpszymon.ps.gz`.

[14] Gusfield, D.: Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology. Cambridge University Press, 1997.

[15] Hunt, J. W.—Szymanski, T. G.: A Fast Algorithm for Computing Longest Common Subsequences. Comm. of the ACM, Vol. 20, 1977, No. 5, pp. 350–353.

[16] Hyyrö, H.: Bit-Parallel LCS-length Computation Revisited. Proc. of the 15th Australasian Workshop on Combinatorial Algorithms (AWOCA), University of Sydney, Australia, 2004.

[17] Lemström, K.—Ukkonen, E.: Including Interval Encoding Into Edit Distance Based Music Comparison and Retrieval. Proc. of Symposium on Creative & Cultural Aspects and Applications of AI & Cognitive Science, 2000, pp. 53–60.

[18] Mäkinen, V.—Navarro, G.—Ukkonen, E.: Transposition Invariant String Matching. J. of Algorithms, Vol. 56, 2005, No. 2, pp. 124–153.

[19] Mehlhorn, K.: Data Structures and Algorithms 1: Sorting and Searching. Springer–Verlag, 1984.

[20] Navarro, G.: Dynamic Dictionaries in Constant Worst-Case Time. Technical Report TR/DCC-2007-11, University of Chile, Department of Computer Science, October 2007, ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/consthash.ps.gz.

[21] Navarro, G.—Grabowski, Sz.—Mäkinen, V.—Deorowicz, S.: Improved Time and Space Complexities for Transposition Invariant String Matching. Technical Report TR/DCC-2005-4, University of Chile, Department of Computer Science, March 2005, ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/mnloglogs.ps.gz.

Sebastian Deorowicz received his M. Sc. degree in Silesian University of Technology in 1998 and Ph. D. degree in the same university in 2003, both in computer science. His research interests are in string matching, sequence alignment, data compression, and combinatorial optimization. He has published about 20 journal and conference papers. He is currently an assistant professor at Silesian University of Technology in Gliwice.

Szymon Grabowski received his M. Sc. degree in Łódź University in 1996 and Ph. D. degree in AGH University of Science and Technology (formerly known as University of Mining and Metallurgy) in Cracow in 2003, both in computer science. His former research, including Ph. D. dissertation, involved nearest neighbor classification methods in pattern recognition, also with applications in image processing. Currently, his main interests are focused in string matching and text indexing algorithms, and data compression. Some of his particular research topics include various approximate string matching problems, compressed text indexes, and XML compression. He has published about 80 papers in journals and conferences. He is currently an assistant professor at Computer Engineering Department of Technical University of Łódź.