Nice to be a chimera: A hybrid algorithm for the longest common transposition-invariant subsequence problem

Szymon Grabowski, Sebastian Deorowicz

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. In this work, we propose a hybrid algorithm picking the better of the two algorithms for individual subproblems. Experiments on music (MIDI) show that the proposed algorithm outperforms the faster of the two component algorithms by a factor of 1.4–1.9, depending on sequence lengths. Also for uniformly random data, the hybrid is the winner if the alphabet is not too large (up to 128 symbols).**

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

I. Introduction

One of the most important problems in the field of string matching concerns the *longest common subsequence* (LCS) of two sequences. In a weaker version, the LCS problem can be stated like that: Given two sequences: $A = a_1, \ldots, a_m$ and $B = b_1, \ldots, b_n$, over an alphabet $\Sigma = \{0, \ldots, \sigma - 1\}$, report the length *l* of the longest subsequence $\langle a_{i_1}, a_{i_2}, \ldots, a_{i_l} \rangle$ of *A*, where $i_k < i_{k+1}$ for all *k*, $a_{i_1} = b_{j_1}, a_{i_2} = b_{j_2}, \ldots, a_{i_l} = b_{j_l}$, and $j_k < j_{k+1}$ for all *k*. A harder version of the problem asks for the sequence itself (which does not have to be unique), not just its length.

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 [1]. Also, numerous variations of the problem have been posed, see, e.g., [2] for details.

One of the LCS variations, introduced relatively recently [3], 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) that we talk about is to find the length of the longest subsequence $\langle a_{i_1}, a_{i_2}, \ldots, a_{i_l} \rangle$ of *A* such that $a_{i_1} = b_{j_1} + t, a_{i_2} = b_{j_2} + t, \ldots, a_{i_l} = b_{j_l} + t$, 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*. This corresponds to a music phrase (melody) shifted to another key, which is perceived by humans as the same melody. The alphabet size in music (MIDI) application is usually 128. As long as this does not lead to confusion, we will denote the length of LCS (LCTS) by LLCS (LLCTS).

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 existing 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 string matching, making use of the simple fact that any real CPU works on several bits in parallel (usually 32 or 64 nowadays).

For LCS, there are known algorithms of $O(mn/w)$ worst case time complexity [5,6,7], 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 $O(mn\sigma/w)$ time complexity, not counting a preprocessing stage. Experiments show [8] that this approach is quite practical.

Sparse dynamic programming (SPD) [9] is a technique of visiting only selected cells of the DP matrix, namely those corresponding 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 [10], 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, subdued to a couple of refinements, allowed to obtain $O(mn \log m)$, $O(mn \log\log m)$ [9], $O(mn \log \sigma)$ [11], and finally $O(mn \log\log \sigma)$ [12,8] time complexity.

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 starts to dominate. Our idea is to use the bit-parallel technique for frequent transpositions and the Hunt–Szymanski algorithm for the rare ones. Experiments in Section VII on MIDI and random data confirm attractiveness of this simple approach.

II. The Hunt–Szymanski algorithm for LCS

A simple idea proposed in 1977 by Hunt and Szymanski [10] has become the departure point for the theoretically best LCS algorithms [13,14], and also for the best LCTS algorithms based on sparse dynamic programming. In this section, we present the HS algorithm in detail.

Szymon Grabowski – Politechnika Lodzka, Katedra Informatyki Stosowanej, al. Politechniki 11, 90-924 Lodz, POLAND. E-mail: sgrabow@kis.p.lodz.pl.

Sebastian Deorowicz – Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16, 44-100 Gliwice, POLAND. E-mail: sebastian.deorowicz@polsl.pl.

We start with a definition. We will say that a cell (i, j) of the dynamic programming matrix *M* stores a match of rank *k* iff $a_i = b_j$ and LLCS($a_1,\ldots,a_i,b_1,\ldots,b_j$) = *k*. Now we can present the algorithm.

Let the matrix *M* have $m+1$ rows and $n+1$ columns. W.l.o.g. we assume $m \le n$. We also assume $\sigma = O(n)$.

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 perform a $O(1)$-time lookup to access the occurrence list of this character in *A*. Traversing a list clearly requires $O(1)$ time per item. For some practical benefits 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 $T[1,m]$, which stores at position *j* the leftmost seen-so-far column with a match of rank *j*.

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 cell 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 $T[t] \ge j$. If there is no such index *t*, that is, $T[h] < j$, for $h = 1,\ldots,k_{\max}$, then we set $T[k_{\max}+1] := j$ 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 yield any update to *T*. If, however, $T[t] > j$, then we set $T[t] := j$, as there hasn't been yet a match with rank *t* in the *j*th (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 us denote the number of all matches in *M* with the symbol *r*. It is easy to notice that the time complexity of the algorithm depends on how fast we 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 cells, even if empty), but since the non-empty range of *T* never has more than l = LLCS(*A*,*B*) elements, it is more precise to express the worst case complexity with $O(n + r\log l)$. Note that we can ignore the preprocessing cost since it is never dominating.

III. The Deorowicz refinement

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 a binary search, in particular, using the van Emde Boas (vEB) dynamic data structure [15] which is applicable if the universe of keys is nicely bounded. In our problem, this translates to $O(n + r\log\log m)$ worst case complexity. The possibility of using the vEB structure was noticed already by Hunt and Szymanski in their original work. There are even better (and more complex) theoretical algorithms [13,14] based on the idea of Hunt-Szymanski, where for example the symbol *r* is replaced with *D*, the number of so-called dominant matches ($D \le r$).

In this section, we are going to present a practical HS variation by Deorowicz [8], which was used in the cited work for calculating LCTS in $O(mn\log\sigma/\log w)$ 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 that we cite was to support the successor queries with a *w*-ary tree, where *w* is the size of the machine word (in bits). More precisely, the *w*-ary tree is a complete tree of arity *w*, storing unique keys from $[0, v-1]$ range, in which each node is an array of exactly *w* bits. The height of this tree is $O(\log v/\log w)$. 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).

In Deorowicz's LCTS algorithm, one *w*-ary tree (Fig. 1, taken from [8]) is used for each transposition, and the invariant is that each tree stores the values of the *T* array for the corresponding transposition. To check if *j* is in the tree, 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*(log *v* / log *w*). 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 appears 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*(log *v* / log *w*). A straightforward solution would take $v = m$, but in the cited work it was shown how to decrease *v* to σ, which is beneficial both for speed and storage occupancy.

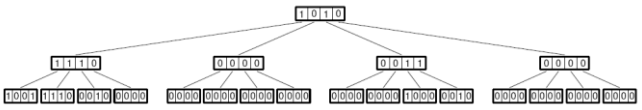

Fig. 1. Sample *w*-ary tree (*w*=4) [8] storing integers from [0, 63]. The integers 0, 3, 4, 5, 6, 10, 40, 46 are stored in leaves.

IV. Bit-parallel approach

Two adjacent values in a row (or in a column) of the matrix *M* differ by at most 1. This simple observation was the starting point of the first bit-parallel LCS solving algorithm, invented by Allison and Dix [5]. 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 $O(\lceil m/w \rceil n)$ and does not depend on the content of *A* and *B* sequences.

Future attempts along these lines, by Crochemore et al. [6] and Hyyrö [7], 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 $PM_\lambda[i]$ 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 [6], which was successively reduced by Hyyrö [7] to four operations, without, e.g., table lookups except for the references to vectors PM_λ .

1. **ComputePM**(A)
2. $V' \leftarrow 1^m$
3. **For** $j \in 1 \ldots n$ **Do**
4. $\quad U \leftarrow V' \;\&\; PM_{B_j}$
5. $\quad V' \leftarrow (V' + U) \mid (V' - U)$
6. **Return the number of unset bits in** V'

Fig. 2. Hyyrö's bit-parallel algorithm [7] for LCS(*A*, *B*) computation. The preprocessing is performed in line 1 (we omit the code for ComputePM() function).

In Hyyrö's experiments, 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 Fig. 2 was faster than the Crochemore et al. by 15%. The test machine in that work was an AMD Athlon64 with *w*=64.

V. From LCS to LCTS

Mäkinen et al. [9] 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 of $2\sigma - 1$ transpositions. The total amount of matches is exactly *mn*, and this easily implies the time complexity of $O(nm\log l)$, or $O(nm\log\log m)$ 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 the previous section, 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.

VI. 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 used a simple criterion for the split between the HS-friendly and BP-friendly transpositions. Namely, we sorted the transpositions by frequency (i.e., number of matches for each individual transposition), and used the cumulative fraction of all matches in those transpositions, as the threshold (more frequent transpositions are submitted to the BP algorithm); the value of this threshold was set experimentally for each dataset (see next section).

VII. 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 X2 5000+ (CPU clock 2600 MHz) machine with 2 GB of RAM, running Windows Vista64 operating system. We have implemented all the algorithms in C++, and compiled with Microsoft Visual C++ 2005™.

We considered two cases: running the algorithms on music data, and running them on uniformly random data, for varying alphabet size.

For the first set of experiments we used a concatenation of 7543 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...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., [8,16]), 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 are the medians over all 101 trials.

Fig. 3 demonstrates the relation between the (percentage) amount of most frequent transpositions and the amount of matches covered by them. The two curves (for *n* = 256 and *n* = 1024) are similar. We can see, for example, that the top 20% of the existent transpositions (sorted by frequency) already

cover at least half of the matches, while 60% of the existent transpositions are enough to cover over 90% of matches.

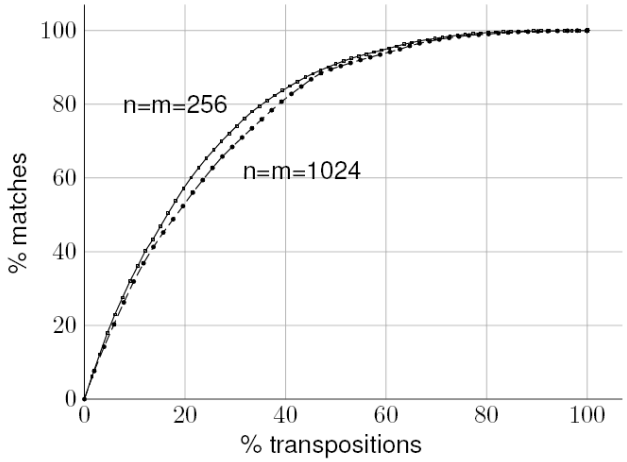

Fig. 3. MUSIC data. Cumulative amount of matches in the transpositions sorted by frequency.

Figs 4–9 show the overall processing time of our hybrid in the function of the percentage of the matches handled by the bit-parallel component. In Figs 4–6, the sequences are taken from the music data, and their length n is set to 256, 1024 and 4096, respectively. Figs 7–9 illustrate the behavior on random data (again, tested sequence lengths of 256, 1024 and 4096), where also the alphabet size was a parameter, from 16 to 256, and apart from that, the test methodology was identical.

As one can see, for the music data, the best split is to have about 80% matches (from the most frequent transpositions) handled by the BP algorithm, while the remaining 20% matches handled by the HS variant. In other words, 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 applied exclusively.

The speedup factor of the hybrid algorithm, using the best threshold for each case, over the better of the two components (i.e., BP) on the music data varies from 1.37 (n = 256) to 1.93 (n = 4096), i.e. improves with growing n. If we use the 80% threshold for all experiment with the music dataset, then the speedup factors drop only slightly, if at all: to 1.34 and 1.93, respectively. Note that we skip non-existent transpositions in the BP algorithm, which boosts its performance on the music data very significantly.

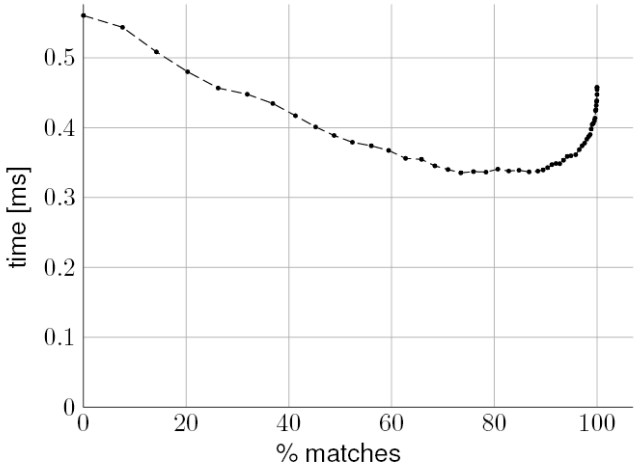

Fig. 4. MUSIC, n = 256. Processing time in function of the % of matches handled by the bit-parallel component.

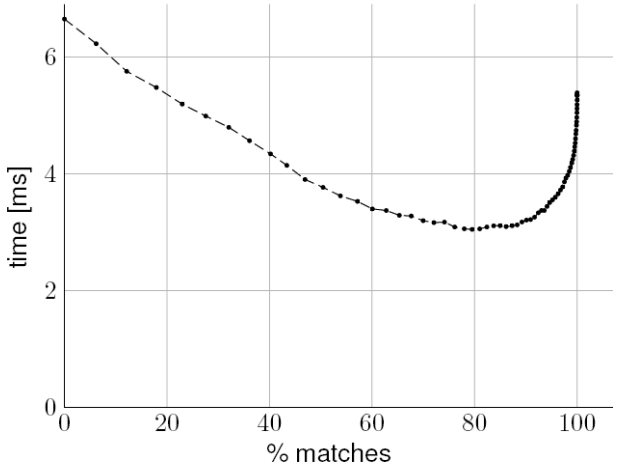

Fig. 5. MUSIC, n = 1024. Processing time in function of the % of matches handled by the bit-parallel component.

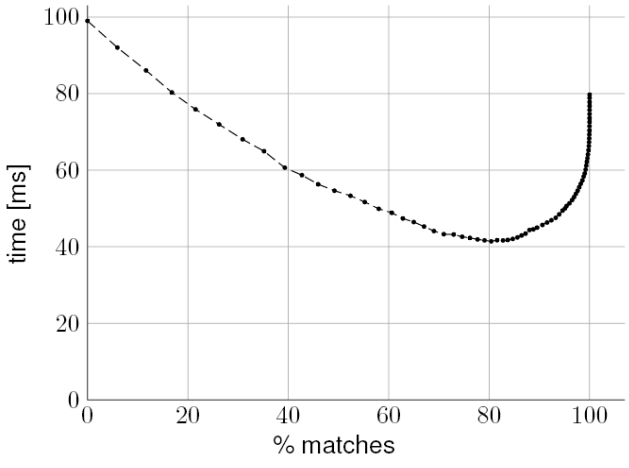

Fig. 6. MUSIC, n = 4096. Processing time in function of the % of matches handled by the bit-parallel component.

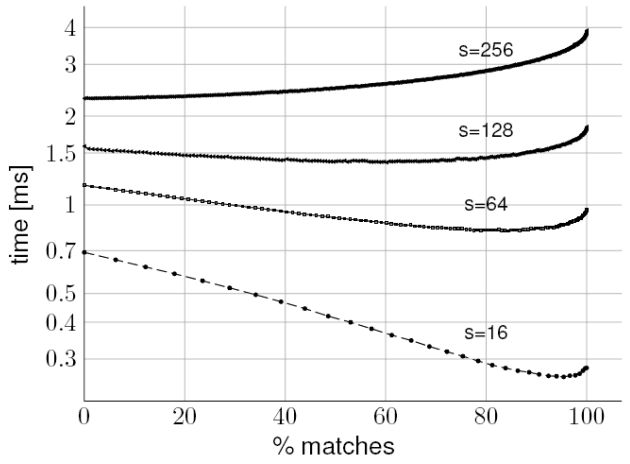

Fig. 7. RANDOM, n = 256. Processing time in function of the % of matches handled by the bit-parallel component.

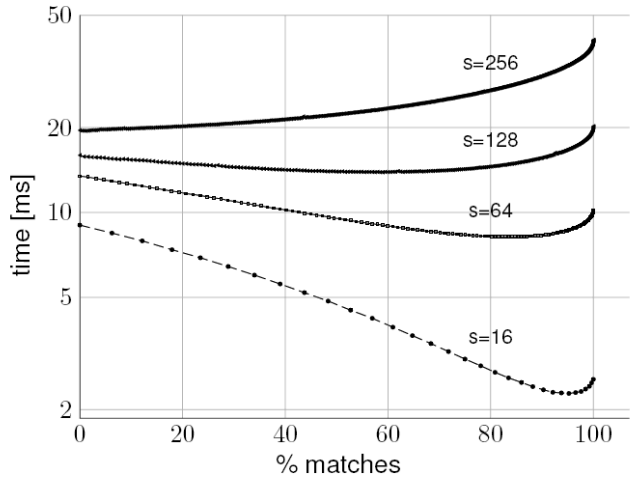

Fig. 8. RANDOM, n = 1024. Processing time in function of the % of matches handled by the bit-parallel component.

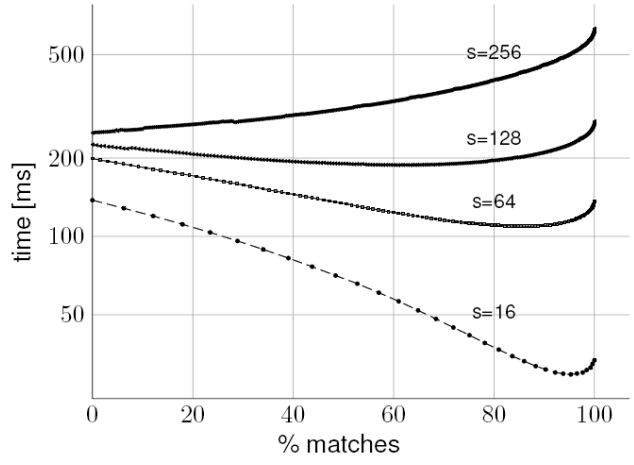

Fig. 9. RANDOM, n = 4096. Processing time in function of the % of matches handled by the bit-parallel component.

On 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 Figs 7–9), even over four times in case of σ = 16 and n = 4096, but the picture changes for σ = 128 and σ = 256. Interestingly, for small alphabets the HS component beats the BP component on some (few) transpositions, so the hybrid, with the optimal threshold, again appears better than both its components (with the speedup of 7–18% over BP), but for a large enough alphabet (σ = 256) the BP algorithm can win on no transposition with the HS algorithm, hence the "optimal" hybrid degenerates into the HS component. The border case is σ = 128 where HS takes the lead but its advantage over BP is quite moderate; in that case the hybrid algorithm is faster than HS by 13–19%. A different threshold should be selected for each alphabet size.

VIII. 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 further work will be focused on finding more elegant ways to separate the domains of individual components. We also intend to repeat the experiment with 64-bit machine words, and we hope to optimize a little more the component algorithm codes.

References

[1] 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), pp. 39–48, 2000.

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

[3] Lemström K., Ukkonen E.: *Including interval encoding into edit distance based music comparison and retrieval*, Proc. of the AISB'2000 Symp. on Creative & Cultural Aspects and Applications of AI & Cognitive Science, pp. 53–60, Birmingham, UK, 2000,

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

[5] Allison L., Dix T.L.: *A bit-string longest common subsequence algorithm*, Information Processing Letters 23(6):305–310, 1986.

[6] Crochemore M., Iliopoulos C.S., Pinzon Y.J., Reid J.F.: *A fast and practical bit-vector algorithm for the longest common subsequence problem*, Proc. of the 11th Australasian Workshop on Combinatorial Algorithms (AWOCA), pp. 75–86, University of Newcastle, NSW, Australia, 2000.

[7] Hyyrö H.: *Bit-Parallel LCS-length Computation Revisited*, Proc. of the 15th Australasian Workshop on Combinatorial Algorithms (AWOCA), pp. 16–27, University of Sydney, Australia, 2004.

[8] Deorowicz, S.: *Speeding up Transposition-Invariant String Matching*, Information Processing Letters, 100(1): 14–20, 2006.

[9] Mäkinen V., Navarro G., Ukkonen E.: *Transposition Invariant String Matching*, J. of Algorithms, 56(2):124–153, 2005.

[10] Hunt J.W., Szymanski T.G.: *A Fast Algorithm for Computing Longest Common Subsequences*, Comm. ACM, 20(5):350–353, 1977.

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

[12] 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. Available at: ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/mnloglogs.ps.gz

[13] Apostolico A., Guerra C.: *The Longest Common Subsequence Problem Revisited*, Algorithmica, 2(1):316–336, 1987.

[14] Eppstein D., Galil Z., Giancarlo R, Italiano G.F.: *Sparse dynamic programming I: linear cost functions*, J. of the ACM, 39(3):519–545, 1992.

[15] van Emde Boas P., Kaas R., Zijlstra E.: *Preserving order in a forest in less than logarithmic time and linear space*, Information Processing Letters, 6(3):80–82, 1977.

[16] Fredriksson K., Mäkinen V., Navarro G.: *Flexible Music Retrieval in Sublinear Time*, International Journal of Foundations of Computer Science, 17(6):1345–1364, 2006.