Speeding up Transposition-Invariant String Matching

Sebastian Deorowicz[1][2]

Silesian University of Technology,
Institute of Computer Science,
44-100 Gliwice, Akademicka 16, Poland

Abstract

Finding the longest common subsequence (LCS) of two given sequences $A = a_0a_1 \ldots a_{m-1}$ and $B = b_0b_1 \ldots b_{n-1}$ is an important and well studied problem. We consider its generalization, transposition-invariant LCS (LCTS), which has recently arisen in the field of music information retrieval. In LCTS, we look for the longest common subsequence between the sequences $A + t = (a_0 + t)(a_1 + t) \ldots (a_{m-1} + t)$, and B where t is some integer. This means that shifting all the symbols in the A sequence by some value is allowed. We present two new algorithms, matching the currently best known complexity $O(mn \log\log \sigma)$, where σ is the alphabet size. Then, we show in the experiments that our algorithms outperform the best ones from literature.

Key words: longest transposition invariant common subsequence, LCS, music information retrieval, transposition invariance

1 Introduction

The problem of finding the longest common subsequence (LCS) of two given sequences A and B is well studied in literature and has applications in many areas. Recently, its generalization, in which we allow to shift the values of one of the sequences by some amount t proved to be more relevant in the music information retrieval [3, 7, 8].

In music analysis, one often wants to compare how similar are two music pieces. The pieces are commonly stored as sequences of pitches and durations. The melody, however, can be recognized from only the pitch sequence and string matching algorithms are popular in this field.

A special property of music is, however, that people usually perceive the same melody even when it is shifted from one key to another. This equals to adding a constant to all pitch values in the sequence. The differences between pitches

[1]E-mail: `Sebastian.Deorowicz@polsl.pl`

[2]Url: `http://www-zo.iinf.polsl.gliwice.pl/\~sdeor`

are more important than their absolute values. Therefore the LCS, the longest string which can be obtained from both sequences by removing some symbols, is a poor candidate as a measure of similarity for music; we should rather deal with transposition invariant version of string matching.

2 Definitions and Background

The sequences $A = a_0a_1 \dots a_{m-1}$ and $b = b_0b_1 \dots b_{n-1}$ are over Σ, where $\Sigma \subset \mathbb{Z}$, called an *alphabet*, is a finite subset of integers. We assume here $\Sigma = \{0, \dots, \sigma\}$, but our algorithm can be easily adopted for any subset (in a similar way like in [12]). A sequence X' is a subsequence of X, when it can be obtained from X by deleting zero or more symbols. A sequence is the *longest common subsequence* of A and B when it is a subsequence of both A and B and has the largest possible length. A transposed copy of the sequence A denoted as $A + t$, for some $t \in \mathbb{Z}$ is $(a_0 + t)(a_1 + t) \dots (a_{m-1} + t)$. The *longest transposition-invariant common subsequence* (LCTS) of A and B is the longest common subsequence of B and $A + t$ for any $-\sigma \leq t \leq \sigma$. Since the problem is symmetric, we can assume without a loss of generality that $m \leq n$. For simplicity of presentation, we denote $x_0x_1 \dots x_k$ by X_k.

When $a_i = b_j$ we say we have a *match* for a pair (i, j), and when $a_i + t = b_j$ we say we have a *t-match* for this pair. If (i, j) is a match and the LCS length for A_i and B_j is k, then k is a *rank* for (i, j). Similarly, if (i, j) is a t-match, we have a t-rank for it.

The easiest way to develop an algorithm for LCTS is to use some classical LCS algorithm for all possible values of t (see [1, 11] for a survey of LCS algorithms). It means, however, that the time complexity is $O(\sigma)$ times greater than the complexity of the base algorithm.

There are also algorithms specialized for LCTS. The reader is referred to the papers by Mäkinen *et al.* [10] and Lemström *et al.* [6] for an extensive description of the existing methods. An algorithm of the worst-case time complexity $O(mn \log \log m)$ introduced in the former paper was recently improved [12] to $O(mn \log \log \sigma)$, which currently is the best known complexity for the problem. Unfortunately, the best algorithms are rather slow. (Some experiments on practical efficiency of the LCTS methods are given by Lemström *et al.* [6].)

3 Our approach

Our proposal for computing the length of LCTS is presented in Figure 1. The algorithm processes the matrix of size $m \times n$ in a similar way as the classical dynamic programming solutions for LCS. The matrix is traversed from top to bottom and within each row from left to right. During the traversal we compute the lengths of the LCS for all the possible values of t. The lengths are stored in the array $L[-\sigma..\sigma]$.

```
{$A[0..m-1]$, $B[0..n-1]$ }
for $i \leftarrow -\sigma$ to $\sigma$ do
    $L[i] \leftarrow 0$; $N[i] \leftarrow -1$;
    for $j \leftarrow 0$ to $(n+\sigma-1)$ div $\sigma - 1$ do $E[i][j]$.init($\sigma$);
for $i \leftarrow 0$ to $m-1$ do
    for $j \leftarrow 0$ to $n-1$ do
        $t \leftarrow B[j] - A[i]$;
        if $j + i \times n > N[t]$ and not $E[t][j$ div $\sigma]$.check($j$ mod $\sigma$) then
            $s \leftarrow E[t][j$ div $\sigma]$.findsucc($j$ mod $\sigma$);
            if $s < 0$ then
                for $k \leftarrow j$ div $\sigma + 1$ to $(n+\sigma-1)$ div $\sigma - 1$ do
                    if $E[t][k]$.min() $\geq 0$ then
                        $s \leftarrow k \times \sigma + E[t][k]$.min(); break;
            else
                $s \leftarrow s + j - j$ mod $\sigma$;
            $E[t][j$ div $\sigma]$.insert($j$ mod $\sigma$);
            if $s \geq 0$ then
                $N[t] \leftarrow s + i \times n$; $E[t][s$ div $\sigma]$.remove($s$ mod $\sigma$);
            else
                $N[t] \leftarrow n - 1 + i \times n$; $L[t] \leftarrow L[t] + 1$;
$lcts \leftarrow L[-\sigma]$;
for $i \leftarrow -\sigma + 1$ to $\sigma$ do $lcts \leftarrow$ max($lcts, L[i]$);
```

Figure 1: An $O(mn \log\log\sigma)$ algorithm for LCTS. (The operations findsucc and min return -1 when there is no successor or minimal value respectively.)

After processing the $(i-1)$th row, the E data structure store the information about the LCS for A_{i-1} and B for all possible values of t. It stores the matches, $(i_1, j_1), (i_2, j_2), \ldots, (i_k, j_k)$ of ranks $1, 2, \ldots, k$, respectively, with the numbers, j_x $(x = 1, 2, \ldots, k)$, of the leftmost possible columns. Processing the current row, i, for each t-match (i, j) we look for a t-match (i_p, j_p) in E, with the nearest possible, but greater than j, column number. It has some t-rank, p, and it is obvious that the t-match (i, j) has the same t-rank. Therefore, we replace the t-match (i_p, j_p) with (i, j) because the latter has the lower column number. Since all the possible t-matches from $(i, j+1)$ to (i, j_p) have the same t-rank as (i, j), we set the value $N[t]$ to $i_p \times n + j_p$ (total number of cells in the above rows and cells left than $(i_p, j_p + 1)$ in the current row) to skip them.

Let us take a look at Figure 2 where the internal states of the data structures are presented for a sample data. (For simplicity, we focus our attention on $t = 0$ only. We assume here also $\sigma = 4$.) After processing the 2nd row, the E data structure stores pairs $(1, 0), (2, 2), (2, 11)$. These are 0-matches of 0-ranks 1, 2, 3 with the

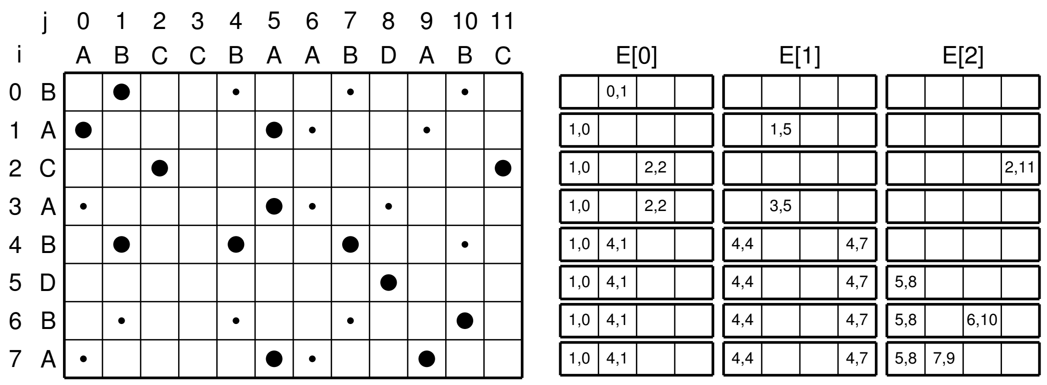

Figure 2: Example of the algorithm in work

lowest possible column numbers. Now, we process the 3rd row. We see a 0-match $(3,0)$, but it does not improve any 0-match, since we already have a 0-match for column 0 (the 0-match is marked as a small dot which indicates it is not processed in lines 08–19). The next 0-match, $(3,5)$, is processed because we actually have no 0-match for column 5 in E (we denote this in the figure using a big dot). We look for the nearest 0-match with a greater column number finding $(2,11)$, and since both 0-matches have the same 0-rank, we remove $(2,11)$ from E and insert $(3,5)$. We also make a note (in array N) that no other 0-match in the current row with a column number lower or equal 11 will have a higher 0-rank. Therefore, for the 0-matches $(3,6)$ and $(3,8)$ we fail in line 07. After processing the 3rd row, we have the 0-matches of 0-ranks 1, 2, 3 with the lowest possible column numbers for A_3 and B. In the end, we have the t-matches for successive ranks for the whole A and B. The t-matches, however, does not form an LCTS, we only know its length. (In fact, it suffices to store only column numbers in E for computing the LCTS length.)

The data structure E is organized as a family of van Emde Boas trees [4, 13], each of size σ. They store the values $[0,\sigma-1]$, $[\sigma,2\sigma-1]$, etc. We always look for the successor of j in the current tree (line 08) and when the tree does not contain it, we browse the successive trees (lines 09–14).

Each van Emde Boas tree stores up to σ different values, so the time complexity for all the basic operations on it is $O(\log\log\sigma)$. The initialization of the L, E, N arrays takes time $O(\sigma)$. The check, findsucc, insert, remove operations are executed at most one time per each execution of the inner loop (lines 05–19) so their contribution to the total time complexity is $O(mn\log\log\sigma)$. The min operation can be executed several times during the single execution of the inner loop. Fortunately, thanks to the existence of the N array, for each execution of the main loop (lines 04–19), for each of $O(\sigma)$ t values, min will be executed only $O(n/\sigma)$ times, which gives $O(mn)$ executions in total. This operation can be implemented in $O(1)$ time, so the final complexity of the algorithm is $O(mn\log\log\sigma)$. Even when $\sigma > nm$ the complexity remains the same, since in this case the initialization can be done in $O(nm)$.

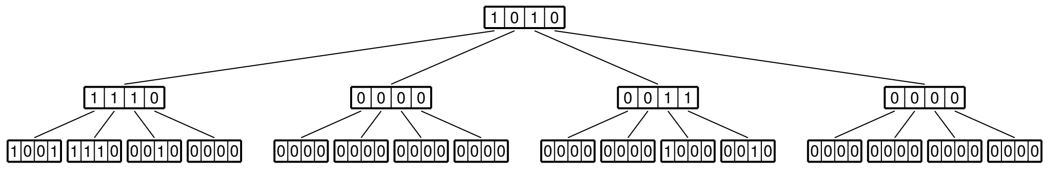

Figure 3: Sample tree of arity 4 storing integers from the range $[0, 63]$ (the integers $0, 3, 4, 5, 6, 10, 40, 46$ are stored in leaves)

The space complexity is dominated by van Emde Boas trees. There is $O(\sigma) \times O(n/\sigma)$ such trees and each of them occupies $O(\sigma)$, which leads to the memory consumption of $O(n\sigma)$. In fact, it suffices to interchange, if necessary, the A and B sequences obtaining $m \geq n$ to get the space complexity $O(\min(m, n)\sigma)$.

The presented algorithm equals the time complexity obtained recently [12], but is much simpler. Since van Emde Boas trees are slow in practice, we propose to replace them with some other data structures. Convenient candidates are w-ary trees, where w is the processor word size. Each inner node stores up to w bits; a bit is set iff the corresponding child node is non-empty. Data are stored in leaves; a bit is set iff the integer is in the tree (see Figure 3 for example).

Assuming a number of leading zero bits in an integer operation works in $O(1)$ time (which is true for modern processors) we can implement all the necessary operations on the tree in time $O(\lceil \log \sigma / \log w \rceil)$. This leads to the $O(mn\lceil \log \sigma / \log w \rceil)$ algorithm for the LCTS problem. The memory occupation remains unchanged.

It is also possible to precompute the number of leading zeros for all the integers from the range $[0, 2^{\lceil w/2 \rceil} - 1]$ in time $O(2^{\lceil w/2 \rceil})$. Thanks to the lookup table we can find the number of leading zeros in every w-bit integer in 2 lookups. This strategy is attractive when $O(2^{\lceil w/2 \rceil}) = O(mn \log \sigma / \log w)$.

4 Comparison of algorithms for LCTS

To compare the LCTS algorithms in practice, we measured their speed on pitch sequences from real MIDI files ($\sigma = 127$). This makes the experiments different from presented by Lemström *et al.* [6] since they used randomly produced sequences over the alphabet $[0, 127]$. In our experiments, $n = m$ and the sequence lengths varied from 20 to 10000. The examined methods are:

- BBB—a binary branch and bound method [6] of complexity $O((mn + \log \sigma)\sigma)$ in the worst case and $O(mn + \log \log \sigma) \log \sigma)$ in the best case,

- CDP—a classical dynamic programming repeated for all $t \in [-\sigma, \sigma]$ of complexity $O(mn\sigma)$ [8],

- KBB—a k-ary ($k = 3$ in our experiments) branch and bound method [6] of complexity $O((mn + \log(\sigma k/(k-1)))\sigma k/(k-1))$,
- KBBCDP—a combination of classical dynamic programming and binary branch and bound algorithms [6] of complexity $O((mn + \log \sigma)\sigma)$ in the worst case and $O(mn + \log\log \sigma) \log \sigma)$ in the best case,
- SDP—a sparse dynamic programming [9] of complexity $O(mn \log m)$,
- YBP—a bit-parallel algorithm [2] of complexity $O(mn\lceil \sigma/w \rceil)$,
- HBP—a bit-parallel LCS algorithm [5] repeated for all possible t values of complexity $O(\lceil m/w \rceil n\sigma)$,
- OUR-1—our $O(mn \log\log \sigma)$ algorithm,
- OUR-2—our $O(mn\lceil \log \sigma / \log w \rceil)$ algorithm.

The experiments were performed on an AMD Athlon 2500 XP+ machine (1800 MHz real CPU clock). The source codes for the BBB, CDP, KBB, KBBCDP, SDP, YBP, HBP algorithms come from their authors. Some of the routines were optimized to achieve better speed and similar optimization level as our implementations. The running times (shown in Table 1 in milliseconds) are median values of 101 executions for different pairs of sequences. The second column gives the median value of the found LCTS lengths.

As we can see, HBP is the fastest algorithm given in literature. This is an LCS method executed $2\sigma + 1$ times for each possible transposition. Our proposal, OUR-2, of complexity $O(mn\lceil \log \sigma / \log w \rceil)$, runs several times faster for short sequences ($0 < n, m < 500$) and about 25% faster for long ones ($1000 < n, m \leq 10000$). It is the fastest method for all the examined sequence lengths.

5 Conclusions

We introduced two algorithms for computing transposition-invariant longest common subsequence length. The first of them achieves the $O(mn \log\log \sigma)$ worst-case time complexity of the currently best algorithm [12], but is significantly simpler. The second algorithm is a variant of the first one. It offers a complexity $O(mn\lceil \log \sigma / \log w \rceil)$, which for typical values $\sigma = 127$, $w = 32$ is also attractive. Its most important asset is, however, its practical speed. It is the fastest algorithm for the whole examined range of sequence lengths. The space complexity of our methods is $O(\min(n, m)\sigma)$ which is usually a bit worse than $O(\sigma^2 + m)$ for the Navarro *et al.* proposal.

Table 1: Experimental comparison of algorithms solving LCTS (times are given in ms). The SDP algorithm could not be evaluated for long sequences due to its huge memory consumption exceeding 512 MB RAM available on the used computer.

$n = m$	|LCTS|	BBB	CDP	KBB	KBBCDP	SDP	YBP	HBP	OUR-1	OUR-2
20	8	0.18	0.57	0.15	0.14	0.09	0.20	0.14	0.61	0.02
40	15	0.58	1.88	0.49	0.62	0.31	0.35	0.43	1.09	0.04
60	23	1.62	3.98	0.95	1.36	0.63	0.50	0.50	1.19	0.06
80	30	2.13	6.88	1.28	1.80	1.25	0.79	0.79	7.34	0.23
100	38	2.89	10.71	2.42	2.59	1.56	1.04	1.15	7.50	0.31
200	76	10.78	41.67	6.57	9.02	7.82	3.34	2.69	32.37	1.11
300	117	23.66	93.71	15.63	20.32	25.00	6.25	5.47	64.45	1.95
400	151	46.10	164.00	28.15	35.90	46.80	9.87	8.33	73.93	3.13
500	172	74.17	260.67	49.50	62.47	78.00	15.60	11.75	80.45	5.45
750	280	156.25	578.00	179.75	128.88	187.00	31.27	24.00	143.80	11.47
1000	373	289.00	1031.00	332.00	234.20	328.00	53.10	40.70	209.40	20.30
1500	569	648.50	2313.00	734.50	500.00	734.00	109.40	81.20	409.40	47.00
2000	704	1172.00	4125.00	1313.00	1078.00	1343.00	190.60	140.60	850.00	84.20
2500	870	1968.00	6453.00	2234.00	1625.00	2094.00	293.80	215.60	1331.20	128.00
3000	1055	2890.00	9266.00	3187.00	2187.00	3031.00	418.80	300.00	1956.20	181.20
4000	1466	4687.00	16500.00	5266.00	3563.00	5250.00	731.20	528.20	3837.40	325.00
5000	1927	6859.00	25797.00	7844.00	5515.00	8187.00	1131.40	815.60	6384.40	506.20
7500	2636	17547.00	59438.00	19735.00	13719.00	—	2500.00	1804.50	14773.00	1156.50
10000	3774	29422.00	110765.00	33469.00	22687.00	—	4476.50	3195.00	26281.00	2062.50

Acknowledgments

The author thanks Szymon Grabowski and Gonzalo Navarro for discussion on the LCTS problem and suggesting improvements. The source codes by Yoan Pinzon and Heikki Hyyrö were also helpful for the experiments.

References

[1] L. Bergroth, H. Hakonen, T. Raita, A Survey of Longest Common Subsequence Algorithms. Proceedings of the 7th International Symposium on String Processing and Information Retrieval (SPIRE 2000) 39–48.

[2] M. Crochemore, C.S. Iliopoulos, Y.J. Pinzon, Speeding-up Hirschberg and Hunt-Szymanski LCS Algorithms. Proceedings of the 8th International Symposium on String Processing and Information Retrieval (SPIRE 2001) 59–67.

[3] T. Crawford, C. Iliopoulos, R. Raman, String matching techniques for musical similarity and melodic recognition. Computing in Musicology 11 (1998) 71–100.

[4] P. van Emde Boas, R. Kaas, E. Zijlstra, Design and implementation of an efficient priority queue. Mathematical Systems Theory 10 (1977) 99–127.

[5] H. Hyyrö, G. Navarro, Bit-parallel Witnesses and their Applications to Approximate String Matching. Algorithmica 41(3) (2005) 203–231.

[6] K. Lemström, G. Navarro, Y. Pinzon, Practical Algorithms for Transposition-Invariant String-Matching. To appear in Journal of Discrete Algorithms (JDA). Preprint at `http://www.dcc.uchile.cl/~gnavarro/ps/jda04.2.ps.gz`.

[7] K. Lemström, J. Tarhio, Searching monophonic patterns within polyphonic sources. Proceedings of Content-Based Multimedia Information Access (RIAO'2000) 1261–1279.

[8] K. Lemström, E. Ukkonen, Including interval encoding into edit distance based music comparison and retrieval. Proceedings of AISB'2000 Symposium on Creative & Cultural Aspects and Applications of AI & Cognitive Science (2000) 53–60.

[9] V. Mäkinen, G. Navarro, E. Ukkonen, Algorithms for transposition invariant string matching. Proceedings of 20th International Symposium on Theoretical Aspects of Computer Science (STACS'2003). Lecture Notes in Computer Science, volume 2607, (2003) 191–202.

[10] V. Mäkinen, G. Navarro, E. Ukkonen, Transposition Invariant String Matching. To appear in Journal of Algorithms. Preprint at `http://www.dcc.uchile.cl/~gnavarro/ps/jofa04.ps.gz`.

[11] G. Navarro, A guided tour to approximate string matching. ACM Computing Surveys 33(1) (2001) 31–88.

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

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