Bit-Parallel Branch&Bound Algorithm for Transposition Invariant LCS

Kjell Lemström,[1] Gonzalo Navarro[2] and Yoan Pinzon[3]

[1] Department of Computer Science, University of Helsinki, Finland
[2] Department of Computer Science, University of Chile
[3] Department of Computer Science, King's College, London, UK, and
Department of computer Science, Autonomous University of Bucaramanga, Colombia

Main Results. We consider the problem of longest common subsequence (LCS) of two given strings in the case where the first may be shifted by some constant (i.e. transposed) to match the second. For this longest common transposition invariant subsequence (LCTS) problem, that has applications for instance in music comparison, we develop a branch and bound algorithm with best case time $O((m^2 + \log\log\sigma)\log\sigma)$ and worst case time $O((m^2 + \log\sigma)\sigma)$, where m and σ are the length of the strings and the number of possible transpositions, respectively. This compares favorably against the $O(\sigma m^2)$ naive algorithm in most cases and, for large m, against the $O(m^2 \log\log m)$ time algorithm of [2].

Technical Details. Let $A = a_1, \ldots, a_n$ and $B = b_1, \ldots, b_m$ be two strings, over a finite numeric alphabet $\Sigma = \{0 \ldots \sigma\}$. A subsequence of string A is obtained by deleting zero, one or several characters of A. The length of the *longest common subsequence* of A and B, denoted $LCS(A, B)$, is the length of the longest string that is a subsequence both of A and B.

The conventional dynamic programming approach computes $LCS(A, B)$ in time $O(mn)$, using a well-known recurrence that can be easily adapted to compute $LCS(A+c, B)$, where $A+c = (a_1+c), \ldots, (a_n+c)$, for some transposition c, where $-\sigma \leq c \leq \sigma$:

$$LCS^c_{i,0} = 0;\ \ LCS^c_{0,j} = 0;$$
$$LCS^c_{i,j} = \textbf{if } a_i + c = b_j \textbf{ then } 1 + LCS^c_{i-1,j-1} \textbf{ else } \max(LCS^c_{i-1,j}, LCS^c_{i,j-1}).$$

Our goal is to compute the length of the *longest common transposition invariant subsequence*,

$$LCTS(A, B) = \max_{c \in -\sigma \ldots \sigma} LCS^c(A, B).$$

Let X denote a subset of transpositions and $LCS^X(A, B)$ be such that a_{i+1} and b_{j+1} match whenever $b_{j+1} - a_{i+1} \in X$. Now, it is easy to see that $LCS^X(A, B) \geq \max_{c \in X} LCS^c(A, B)$, so $LCS^X(A, B)$ may not contain the actual maximum $LCS^c(A, B)$ for $c \in X$ but gives an upper bound. Our aim is to find the maximum $LCS^c(A, B)$ value by successive approximations.

We form a binary tree whose nodes have the form $[I, J]$ and represent the range of transpositions $X = \{I \ldots J\}$. The root is $[-\sigma, \sigma]$. The leaves have the

form $[c,c]$. Every internal node $[I,J]$ has two children $[I, \lfloor (I+J)/2 \rfloor]$ and $[\lfloor (I+J)/2 \rfloor + 1, J]$.

The hierarchy is used to upper bound the $LCS^c(A,B)$ values. For every node $[I,J]$ of the tree, if we compute $LCS^{\{I \ldots J\}}(A,B)$, the result is an upper bound to $LCS^c(A,B)$ for any $I \leq c \leq J$. Moreover, $LCS^X(A,B)$ is easily computed in $O(mn)$ time if $X = \{I \ldots J\}$ is a continuous range of values:

$$LCS^X_{i,0} = 0; \;\; LCS^X_{0,j} = 0;$$
$$LCS^X_{i,j} = \textbf{if } b_j - a_i \in X \textbf{ then } 1 + LCS^X_{i-1,j-1} \textbf{ else } \max(LCS^c_{i-1,j}, LCS^c_{i,j-1}).$$

We already know that the LCS value of the root is $\min(m,n)$, since every pair of characters match. The idea is now to compute its two children, and continue with the most promising one (higher LCS^X upper bound). For this most promising one, we compute its two children, and so on. At any moment, we have a set of subtrees to consider, each one with its own upper bound on the leaves it contains. At every step of the algorithm, we take the most promising subtree, compute its two children, and add them to the set of subtrees under consideration. If the most promising subtree turns out to be a leaf node $[c,c]$, then the upper bound value is indeed the exact LCS^c value. At this point we can stop the process, because all the upper bounds of the remaining subtrees are smaller or equal than the actual LCS^c value we have obtained. So we are sure of having obtained the highest value.

For the analysis, we have a best case of $\log_2(2\sigma+1) = O(\log \sigma)$ iterations and a worst case of $2(2\sigma+1) - 1 = 4\sigma + 1 = O(\sigma)$ until we obtain the first leaf element. Our priority queue, which performs operations in logarithmic time, contains $O(\log \sigma)$ elements in the best case and $O(\sigma)$ in the worst case. Hence every iteration of the algorithm takes $O(m^2 + \log\log\sigma)$ at best and $O(m^2 + \log\sigma)$ at worst. This gives an overall best case complexity of $O((m^2 + \log\log\sigma)\log\sigma)$ and $O((m^2 + \log\sigma)\sigma)$ for the worst case. The worst case is not worse than the naive algorithm for $m = \Omega(\sqrt{\log\sigma})$, which is the case in practice.

By using bit-parallel techniques that perform several LCS^X computations at the same time [1], the algorithm can be extended to use a t-ary tree.

This technique can be applied also to any distance d satisfying $\min_{c \in X} d^c(A,B) \leq d^X(A,B)$, where $d^X(A,B)$ is computed by considering that a_{i+1} and b_{j+1} match whenever $b_{j+1} - a_{i+1} \in X$. This includes δ-LCS, general weighted edit distance, polyphony, etc., so it enjoys of more generality than most of the previous approaches. It cannot, however, be easily converted into a search algorithm.

References

1. K. Lemström and G. Navarro. Flexible and efficient bit-parallel techniques for transposition invariant approximate matching in music retrieval. In *Proc. SPIRE'03*, LNCS 2857, pp. 224–237, 2003.
2. V. Mäkinen, G. Navarro, and E. Ukkonen. Algorithms for transposition invariant string matching. In *Proc. STACS'03*, LNCS 2607, pp. 191–202, 2003.