Fast algorithm for constrained longest common subsequence problem

Sebastian Deorowicz

December 21, 2005

Abstract

The problem of finding the constrained longest common subsequence (CLCS) for the sequences A and B with respect to the sequence P was introduced recently. The best known algorithms for its solving requires time of order of a product of the sequences length. We introduce a novel approach in which time and memory complexities depends on the number of matches between A, B, and P. The time complexity is never worse than the one of the known algorithms, but typically is better. Experiments show that our method is faster than the known ones and requires less memory.

1 Introduction and background

The problem of finding the longest common subsequence (LCS) of two given sequences A and B is well studied [2, 3, 6] and has a lot of applications in various fields, not only those related to computer science.

There are also a number of problems related to LCS, such as a constrained longest common subsequence (CLCS) introduced recently [8].

1.1 Terms and definitions

Let us assume we have three sequences: $A = a_1, a_2, \ldots, a_n$, $B = b_1, b_2, \ldots, b_m$, and $P = p_1, p_2, \ldots, p_r$. Each of them is composed of symbols over an alphabet Σ of size σ. A sequence X' is a subsequence of X when it can be obtained from X by removing zero or more symbols.

The LCS problem is to find the sequence $L = \mathrm{LCS}(A, B)$ such that L is a subsequence of both A and B, and L has the maximal possible length. The CLCS problem is a generalization of LCS since we have an additional requirement that P has to be a subsequence of L, i.e., $P = \mathrm{LCS}(P, \mathrm{LCS}(A, B))$.

Since the problem is symmetric, we can assume without a loss of generality that $m \leq n$. It is also necessary to assume $r \leq m$, for P cannot be longer than A and B. We say that we have a *match* for the pair (i, j) iff $a_i = b_j$. We call a match *strong* iff for the triple (i, j, k) for which we have a match, i.e., $a_i = b_j$, holds also $a_i = p_k$. For simplicity we use the notation X_s for $x_1, x_2, \ldots, x_s$.

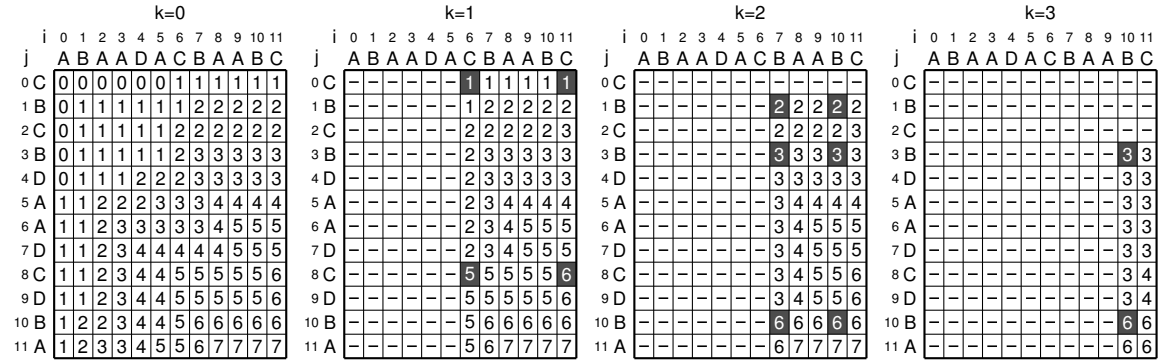


Figure 1: Example of the algorithm by Chin *et al.* for $A = \mathsf{ABAADACBAABC}$, $B = \mathsf{CBCBDAADCDBA}$, $P = \mathsf{CBB}$. (Grayed cells denotes strong matches and '–' symbols denotes $-\infty$ values.)

1.2 Existing methods

Tsai in his paper [8] introduced the CLCS problem and presented an algorithm based on dynamic programming running in $O(m^2n^2r)$ time and $O(mnr)$ space. Recently Peng [7], Arslan and Eğecioğlu [1], and Chin *et al.* [4] showed that the problem is in fact simpler. The authors presented similar algorithms working in $O(mnr)$ time (in both worst and average-case). The algorithms use $O(mnr)$ memory, but Chin *et al.* show that their method can be modified using Hirschberg way [5] to obtain only $O(mr)$ memory complexity while the time complexity is preserved.

2 Proposed algorithms

We start the presentation of our algorithm from the recursive equation by Chin *et al.* [4] (an example of the algorithm is shown in Figure 1), which they introduced to calculate the CLCS:

$$M(i,j,k) = \begin{cases} 1 + M(i-1, j-1, k-1) & \text{if } i,j,k > 0 \text{ and } a_i = b_j = p_k, \\ 1 + M(i-1, j-1, k) & \text{if } i,j > 0, a_i = b_j \text{ and } (k = 0 \text{ or } a_i \neq p_k), \\ \max(M(i-1,j,k), M(i,j-1,k)) & \text{if } i,j > 0 \text{ and } a_i \neq b_j \end{cases} \tag{1}$$

The initialization of the array if done by setting:

$$M(i,0,0) = M(0,j,0) = 0 \text{ and } M(0,j,k) = M(i,0,k) = -\infty, \tag{2}$$

for $i = 0, 1, \ldots, n$, $j = 0, 1, \ldots m$, and $k = 1, 2, \ldots, r$.

There are some interesting properties of the matrix M.

Observation 1. *$M(i,j,k_1) \geq M(i,j,k_2)$ if $k_1 < k_2$.*

Proof. From the initialization of the matrix (Equation 2), it is obviously true for (i,j), where at least one variable of i, j is zero. For $i > 0$ and $j > 0$, the cell $M(i,j,k_1)$ stores the length of the CLCS for A_i and B_j while the sequence P_{k_1} is constrained. The cell $M(i,j,k_2)$ stores the CLCS length while a longer sequence, P_{k_2}, is constrained. Each sequence constraining P_{k_2} obviously constrains also the sequence P_{k_1} which is a subsequence of P_{k_2}, so the length $M(i,j,k_1)$ cannot be smaller than $M(i,j,k_2)$. □

Observation 2. *$M(i_1, j_1, k) \leq M(i_2, j_2, k)$ if $i_1 \leq i_2, j_1 \leq j_2$.*

Proof. The rule of calculation of the M array guarantees that if we have not any strong matches, the value of $M(i_2, j_2, k)$ cannot be smaller than any other value $M(i_1, j_1, k)$ where $i_1 \leq i_2$ and $j_1 \leq j_2$, because the values does not decrease when any of the indexes i, j grows. If we meet a strong match for some triple (i, j, k), we calculate $M(i, j, k)$ as $M(i-1, j-1, k-1)+1$, which is not lower than $M(i-1, j-1, k)+1$ (Observation 1). □

Observation 3. *$M(i_1, j_1, k) < M(i_2, j_2, k)$ if $i_1 < i_2, j_1 < j_2, a_{i_2} = b_{j_2}$.*

Proof. If we have a non-strong match for (i_2, j_2), we calculate $M(i_2, j_2, k)$ as $M(i_2 - 1, j_2 - 1, k) + 1$, so it is obvious that the observation is true, because from Observation 2 $M(i_2 - 1, j_2 - 1, k) \geq M(i_1, j_1, k)$. For a strong match (i_2, j_2), the value of $M(i_2 - 1, j_2 - 1, k-1)+1$ cannot be lower than $M(i_2-1, j_2-1, k)+1$ (Observation 1), so the observation is also true. □

From the above observations and Equation 1 we can write:

$$M(i,j,k) = \begin{cases} \max\limits_{\substack{0 \leq i' < i \\ 0 \leq j' < j}} M(i', j', k-1) + 1 & \text{if } i,j,k > 0 \text{ and } a_i = b_j = p_k, \\ \max\limits_{\substack{0 \leq i' < i \\ 0 \leq j' < j}} M(i', j', k) + 1 & \text{if } i,j > 0, a_i = b_j \text{ and } (k = 0 \text{ or } a_i \neq p_k), \\ \max\limits_{\substack{0 \leq i' \leq i \\ 0 \leq j' \leq j \\ (i' \neq i \vee j' \neq j)}} M(i', j', k) & \text{if } i,j > 0 \text{ and } a_i \neq b_j. \end{cases} \tag{3}$$

To calculate properly the values of M we can restrict to triples (i, j, k) for which we have a match, i.e., $a_i = b_j$, since for non-match triples we make a copy of some cell, without increasing the value stored in it. Therefore we can write:

$$M(i,j,k) = \begin{cases} \max\limits_{\substack{0 \leq i' < i \\ 0 \leq j' < j \\ a_{i'} = b_{j'}}} M(i', j', k-1) + 1 & \text{if } i,j,k > 0 \text{ and } a_i = b_j = p_k, \\ \max\limits_{\substack{0 \leq i' < i \\ 0 \leq j' < j \\ a_{i'} = b_{j'}}} M(i', j', k) + 1 & \text{if } i,j > 0, a_i = b_j \text{ and } (k = 0 \text{ or } a_i \neq p_k). \end{cases} \tag{4}$$

Before we introduce our algorithm for the CLCS problem, let us first consider how to efficiently calculate the following rule for such i and j that $a_i = b_j$, which is similar to the one presented in Equation 4:

$$M^*(i,j) = \max_{\substack{0 \leq i' < i \\ 0 \leq j' < j \\ a_{i'} = b_{j'}}} M^*(i', j') + \alpha(i,j), \tag{5}$$

where $\alpha(i, j)$ is a positive integer for all match pairs i, j, and the boundary conditions are: $M^*(i, 0) = 0$, $M^*(0, j) = 0$ for $0 \leq i \leq n$ and $0 \leq j \leq m$. (When $\alpha(i, j) = 1$ for all pairs (i, j) for which we have a match, then the problem is equivalent to finding the LCS(A, B).)

We process the matrix in a row-wise and in each row in a column-wise matter. We maintain two lists: L^0 and L^1 storing pairs: $\langle column\_no, rank \rangle$ (*rank* is the value stored in M^*). When we start processing the ith row, the list L^0 stores pairs describing the

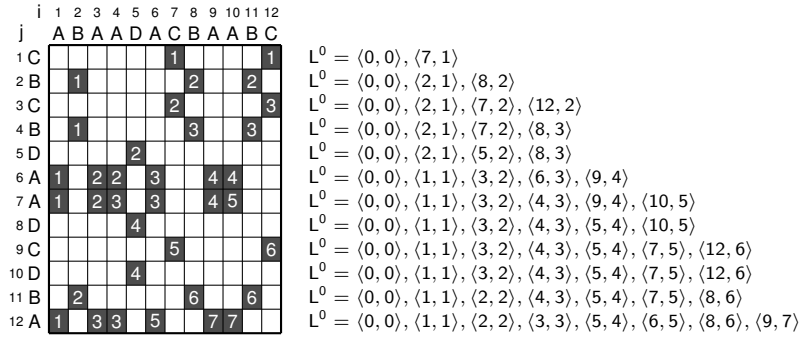


Figure 2: Example of the algorithm calculating M^* matrix for $A = \mathsf{ABAADACBAABC}$, $B = \mathsf{CBCBDAADCDBA}$, and $\alpha(i,j) = 1$ when $a_i = b_j$ and 0 otherwise. The contents of the list L^0 is presented after finishing the row in which the list is placed.

positions in the matrix $M^*[0..i-1, 0..m]$, with the lowest possible column numbers for all the existing rank values. The list is initialized with the pair $\langle 0, 0\rangle$ and is sorted according to increasing ranks.

Now we process the ith row as follows (we visit only matches in this row). For each match, we browse the list L^0 to find the first pair with the higher or equal column number (if no such a pair exist, we stop one position after the last in the list). The previous position on the list stores the pair with the highest rank from all the existing ones in M^* matrix in columns lower than j and rows lower than i. This is in fact the maximal value of $M^*(i', j')$ for all $i' < i$ and $j' < j$, which we need to calculate $M^*(i, j)$. We also make a copy of all the pairs in L^0 at lower positions than the current one to the list L^1, and put the new pair describing the (i, j) match to the list L^1. Then we skip all the pairs of L^0 with not higher ranks than the last added to L^1.

After finishing one row, we copy all the pairs from L^0 to L^1 with ranks higher than the last added to L^1. Then we exchange the lists L^0 and L^1, empty the list L^1, and start processing the next row.

Please confer Figure 2 for an example showing the contents of the L^0 list during the row-wise processing of the M^* matrix to compute the $\mathrm{LCS}(A, B)$.

The length of the lists L^0 and L^1 is bounded by the highest rank of the M^* cells plus one, $l+1$. (When we compute LCS, l is the length of $\mathrm{LCS}(A, B)$.) For each of d matches between A and B, we make $O(1)$ operations and move through the list L^0 (possibly at long distance). The list L^0 is however traversed only m times and we never go backward. Therefore, the total time complexity of the presented procedure is $O(ml+d)$. (We neglect here the cost of initialization of the data structures, e.g., finding the set of d matches, which costs $O(n)$.)

To show that the calculation of the M matrix could be done in a similar way, let us first introduce a second matrix, T:

$$T(i,j,k) = \max_{\substack{0 \le i' < i \\ 0 \le j' < j \\ a_{i'} = b_{j'}}} M(i', j', k) + 1. \tag{6}$$

It stores in $T(i, j, k-1)$ the value used as $M(i, j, k)$ when a match is strong.

We process the M and T matrices in a level-wise matter, i.e., at the beginning we determine the M and T matrices for all the cells for $k = 0$, then we do the same for $k = 1, 2, \ldots, r$. The calculation of the cell values is as follows. If the current match, (i, j, k), is not strong, i.e, $a_i \neq p_k$, then we calculate the values of the $M(i, j, k)$ and

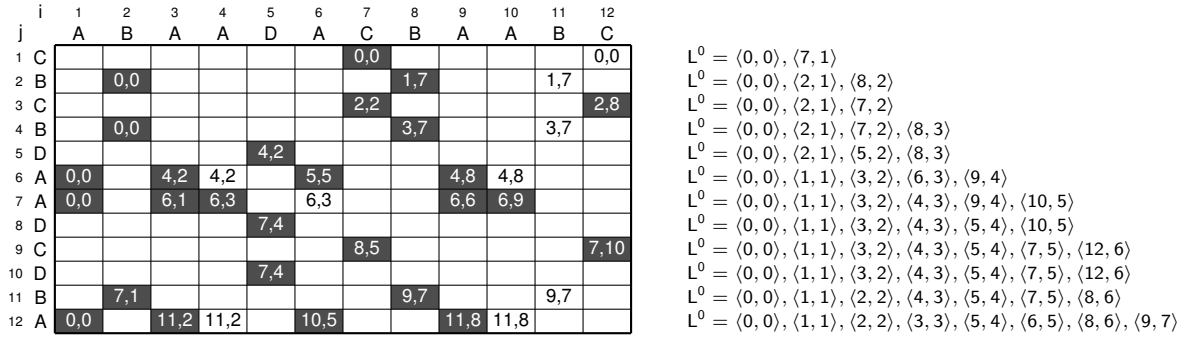


Figure 3: Example of the algorithm calculating the F matrix for $A = \mathsf{ABAADACBAABC}$, $B = \mathsf{CBCBDAADCDBA}$. (For simplicity only 0-level is shown. The grayed cells denotes matches added to the L^1 list.)

$T(i, j, k)$ cells and update the L^0 and L^1 lists in the same way like for the M^* matrix. If the match is strong, we use the stored value of $T(i, j, k-1)$ to find the value of the $M(i, j, k)$ cell, and use the same mechanism like we did for $M^*(i, j)$ to determine the value of $T(i, j, k)$. Then we traverse the L^0 and update the L^1 list with the $M(i, j, k)$ value.

After filling out the whole matrix M, we can find the highest value stored in it, being the length of the CLCS. Alas, there is no simple way to find out the CLCS, not only its length. To make it possible, we need one more three-dimensional matrix, F, which stores in each cell (i, j, k) the pointer to the cell used to calculate the $T(i, j, k)$ value, i.e., the cell with the maximal value from $M(i', j', k)$, where $i' < i$ and $j' < j$. To find out the CLCS, we start from the cell of M with the maximal value, set $k = r$ and do the following loop. If we have a strong match, i.e., $a_i = p_k$, we use the $F(i, j, k-1)$ pointer to set i, j, k variables. If we have a non-strong match, i.e., $a_i \neq p_k$, we use the $F(i, j, k)$ pointer to select the next visited The values of a_i for the visited cells form the CLCS. (Figure 3 shows the contents of the F matrix for the level 0.)

3 Algorithm's complexity and implementation details

There are three three-dimensional matrices and some other lists, which may look as an important disadvantage comparing to the classical algorithm by Chin *et al.* [4], which needs only one three-dimensional matrix. We, however, introduced most of them only for a clear presentation. The simplest optimization is to notice that we do not need the whole T matrix. We need only to store the current and the previous level. The lists L^0 and L^1 together with the previous level of the matrix T store all the necessary information to calculate the values of $M(i, j, k)$ and $T(i, j, k)$. Therefore, we can resign from the M matrix. All the matches between A and B can be determined easily and we only need to precalculate a vector storing the number of appearance of all the alphabet symbols in A and the matrix with the positions of the symbols' appearance. Both can be implemented in $O(\max(m, \sigma))$ memory. The two-dimensional array T can be implemented in $O(d)$ memory for only d cells of it is used. The lists L^0 and L^1 obviously occupy $O(l)$ memory, where l is the length of $\mathrm{LCS}(A, B)$. The F matrix can be implemented in $O(rd)$ memory. The total memory consumption is therefore $O(rd)$.

The initialization of the vector and match-array can be done in $O(\max(n, \sigma))$ time, which is negligible if σ is reasonable. (The case when σ is large will be discussed in the next paragraph.) The processing of each level consumes $O(ml + d)$ time, since this is a similar procedure as discussed for the M^* matrix. Therefore, finding the CLCS needs $O(r(ml + d) + n)$ time, which is better than $O(mnr)$ for the currently known algorithms.

Up to now, we neglected the size of the alphabet, σ. If it is reasonable (comparable to n) the above calculations are true. If σ is large, we should change the algorithm a bit. Instead of storing the number of matches of alphabet symbols in a vector, we use a binary search tree to store the symbols existing in A. When we need to access a list of matches in the current row we search the BST (only one time for each row). This adds a term $O(m \log n + n \log n) = O(n \log n)$ to the time complexity. We can alternatively sort the symbols forming A, remove repetitions and relabel the symbols in both A and B. This approach leads to the additional time complexity term $O(m + n \log n) = O(n \log n)$. In both cases we achieve an additional term $O(n \log n)$ to the time complexity, but it is usually negligible if compared to $O(r(ml + d) + n)$.

The pseudo-code for the calculation of the length of the CLCS is shown in Figure 4. In lines 01–04, we compute the positions of all the alphabet symbols in the sequence A. There are exactly n such positions, so the array *n_pos* can be implemented in $O(n)$ memory. Lines 06–34 contain the main loop, in which we traverse the matrices in a level-wise matter. The initialization of the M matrix is here simplified, and we only set the pseudo-match at position $(0, 0, k)$ to be 0 when we process the 0-level and $-\infty$, when the level is higher. The pseudo-match guarantees that when we stop searching the list L^0, we always have at previous position the match with the highest rank lower than the current one. In the loop 13–34, we browse the current level row-wise. For each row, after copying the pseudo-match, we move through all the row matches. In lines 16–22, we make a copy of the part of the list L^0 to L^1. Then, we add a new match to the list L^1 (lines 23–29). In lines 30–33, we make a copy of the remaining elements from the list L^0. Then, we exchange the list L^0 and L^1 to make the newly created list the current one for the next row, and empty the other list.

4 Evaluation of the algorithms' performance

The existing algorithms computing the CLCS in $O(mnr)$ time are closely related so we selected the one by Chin *et al.* [4] for practical experiments. The algorithm by Tsai [8] of complexity $O(m^2n^2r)$ is obviously the loser so we resign from it in experiments. Therefore, there are only two methods to compare: the one by Chin *et al.* and our. We implemented both algorithms in ANSI C++, compiled using MINGW C++ 3.4.2 compiler, and run on a computer equipped with AMD Athlon XP 2500+ processor (real 1800 MHz CPU clock).

The algorithms' relative complexity does not depend on the length of the constrained sequence, which was confirmed by preliminary experiments, so we selected only one sample value $r = 16$. The complexity of our proposal depends, however, on the number of matches between A and B, and the length of the LCS(A, B). This values depend of course on the alphabet size. Therefore, we compared the algorithms for various values of σ. In the experiments, we used sequences of various lengths. The sequences were generated randomly using the uniform probability distribution of alphabet symbol occurrence.

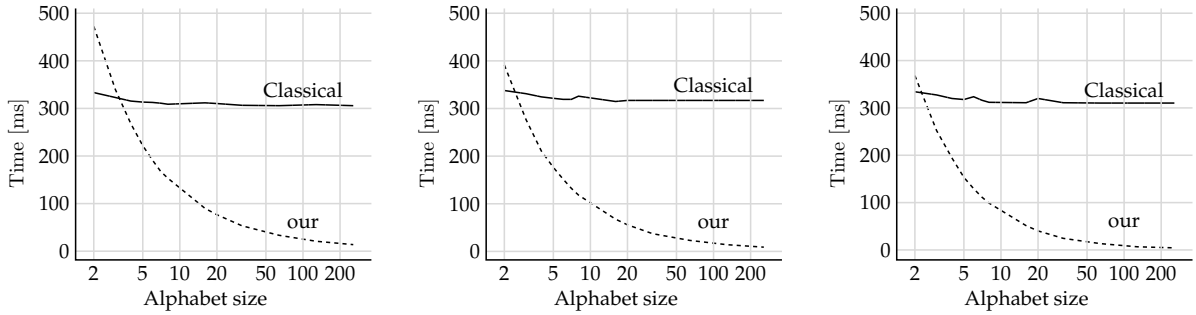
Since, algorithms' relative performance is almost independent on the length of se-

```
    {A[1..n], B[1..m], P[1..k]}
01  for i ← 1 to σ do
02      n_pos[i] ← 0;
03  for i ← 1 to n do
04      pos[A[i]][n_pos[A[i]] ← i; n_pos[A[i]] ← n_pos[A[i]] + 1;
06  for k ← 0 to r do
07      if k = 0 then
08          L^0[0].len ← 0;
09      else
10          L^0[0].len ← −∞;
11      L^0[0].i ← −∞; L^0[0].j ← −∞;
12      N^0 ← 1;  N^1 ← 0;
13      for j ← 1 to m do
14          L^1[N^1] ← L^0[0];  N^1 ← N^1 + 1; p ← 1;
15          for s ← 0 to n_pos[B[j]] − 1 do
16              i ← pos[B[j]][s];
17              while p < N^0 do
18                  if L^0[p].i ≥ i then
19                      break;
20                  else if L^1[N^1 − 1].len < L^0[p].len and L^0[p].len > 0 then
21                      L^1[N^1] ← L^0[p]; N^1 ← N^1 + 1;
22                  p ← p + 1;
23              if k > 0 and B[j] = P[k] then
24                  v ← T[j][i]; T[j][i] ← L^0[p − 1].len + 1;
25              else
26                  v ← L^0[p − 1].len + 1; T[j][i] ← v;
27              F[k][j][i].i ← L^0[p − 1].i; F[k][j][i].j ← L^0[p − 1].j;
28              if L^1[n^1].len < v then
29                  L^1[N^1].i ← i; L^1[N^1].j ← j; L^1[N^1].len ← v; N^1 ← N^1 + 1;
30          while p < N^0 and L^1[N^1 − 1].len ≥ L^0[p].len do
31              p ← p + 1;
32          while p < N^0 do
33              L^1[N^1] ← L^0[p]; N^1 ← N^1; p ← p + 1;
34          L^0 ← L^1; N^0 ← N^1; N^1 ← 0;
35  return L^0[N^0 − 1].len;
```

Figure 4: Algorithm preparing data structures to construct CLCS. Returns length of the CLCS.

```
    {A[1..n], B[1..m], P[1..k]}
01  i ← L^0[N^0].i; j ← L^0[N^0].j; k ← r;
02  for len ← L^0[N^0 − 1].len downto 1 do
03      R[len] ← B[j];
04      if k > 0 and Y[j] = P[k] then
05          k ← k − 1;
06      i' ← F[k][j][i].i; j' ← F[k][j][i].j;
07      i ← i'; j ← j';
08  return R;
```

Figure 5: Algorithm returning the CLCS



$n = 1024$, $m = 1024$, $r = 16$ $n = 2048$, $m = 512$, $r = 16$ $n = 4096$, $m = 256$, $r = 16$

Figure 6: Comparison of the examined algorithms for changing alphabet size.

quences A and B when the factor n/m is constant, we selected three different pairs of lengths n and m. Figure 6 shows the results of experiments for $2 \leq \sigma \leq 256$. Each time is a median of 201 executions.

We can notice that only for the smallest alphabet sizes ($\sigma = 2, 3$) the classical algorithm is faster for the case when $n = m = 1024$. For the other two cases, it is faster only for a binary alphabet. For a typical case, when $\sigma = 4$ (DNA alphabet), our algorithm is about 15–40% faster. It speeds up when the alphabet size grows, because the length of the LCS(A, B) comes shorter, and (more important) the number of matches between A and B decreases. The speed of the classical algorithm also improves a little, while the alphabet size grows. This is caused by the fact that when we have a strong-match (more frequent for small alphabets), we need to take a look at the cell from the previous level, which is located in memory at long distance (and probably were removed from the cache memory). Nevertheless, the improvement of speed for this algorithm is unimportant.

The memory requirements of the classical algorithm is steady and for the case $n = m = 1024$, $r = 16$, the algorithm needs about 72.15 MB. Space requirements of our method depend on the number of matches between A and B. In the experiments, the memory occupation decreases from 70.03 MB for a binary alphabet, through 30.07 MB for the case $\sigma = 4$ and 7.13 MB for $\sigma = 20$ to 0.65 MB for $\sigma = 256$. (We did not experiment with the Hirschberg-like memory optimization of the algorithm by Chin *et al.* since the time necessary to reduce the memory complexity is significant and the algorithm would be much slower than the classical one.)

5 Conclusions

We considered the constrained longest common subsequence problem (CLCS). The problem is rather new and only a few dynamic programming-based algorithms were presented so far. To the best of our knowledge, the best known ones are of $O(mnr)$ time complexity, where the factors are sequence lengths.

We proposed a novel approach, in which the complexity depends on not only the sequences length, but also on the number of matches between the two main sequences. The time complexity of the presented method is $O(r(ml + d) + n)$, where l is the length of the LCS(A, B) and d is the number of matches between A and B. The memory requirements of our algorithm is $O(dr)$, which is also attractive.

Experiments confirmed that the proposed method is faster than the known algorithms

for all practical alphabets (of size not smaller than 4).

Acknowledgments

The author thanks Szymon Grabowski for his comments on the paper contents.

References

[1] A.N. Arslan, Ö. Eğecioğlu. Algorithms for the constrained longest common subsequence problems. Proceedings of the Prague Stringology Conference'04, M. Šimánek, J. Holub Editors, pages 24–32, 2004.

[2] A. Apostolico. General pattern matchings. Chapter in Handbook of Algorithms and Theory of Computation, M.J. Atallah (Editor), Chapter 13, 1998.

[3] L. Bergroth, H. Hakonen, and T. Raita. A survey of longest common subsequence algorithms. In Proceedings of 7th International Symposium on String Processing Information Retrieval (SPIRE), Curuña, Spain, pages 39–48, 2000.

[4] F.Y.L. Chin, A. De Santis, A.L. Ferrara, N.L. Ho, and S.K. Kim. A simple algorithm for the constrained sequence problems. Information Processing Letters, 90:175–179, 2004.

[5] D.S. Hirschberg. Algorithms for the longest common subsequence problem. Journal of the ACM, 24:664–675, 1977.

[6] G. Navarro. A Guided Tour to Approximate String Matching. ACM Computing Surveys, 33(1):31–88, 2001.

[7] Ch.-L. Peng. An Approach for Solving the Constrained Longest Common Subsequence Problem. Master's Thesis, Department of Computer Science and Engineering, National Sun Yat-sen University, Taiwan, 2003. http://etd.lib.nsysu.edu.tw/ETD-db/ETD-search/getfile?URN=etd-0828103-125439&filename=etd-0828103-125439.pdf

[8] Y.-T. Tsai. The constrained common subsequence problem. Information Processing Letters, 88:173–176, 2003.