Case Study of a Learning Algorithm for the Longest Common Subsequence Problem

Eric A. Breimer and Mark K. Goldberg
Rensselaer Polytechnic Institute

Abstract

Based on the behavior of a supervisor-algorithm, we present a learning algorithm which designs an improved algorithm for solving the Longest Common Subsequence problem (LCS). The supervisor is the standard dynamic programming algorithm (DP) for LCS. The learning algorithm applies DP to inputs generated randomly according to a probability distribution $\mathcal{D}$. The optimal solutions generated by DP are used to build a search area. The search area is then used to develop a new algorithm tailored for solving the LCS problem for inputs generated according to $\mathcal{D}$.

We present experiments showing the learning curve of the algorithm and the performance parameters of the new LCS-algorithm after a prescribed approximation ratio has being achieved. In particular, our experiments with two random 0,1-sequences of length n suggest that applying $O(n^{0.630})$ training samples guarantees an LCS-algorithm whose approximation ratio is 0.95 and the running time is $O(n^{1.654})$ (vs the quadratic time of DP). Our experiments also indicate a relationship between the distribution of the input symbols and the structure of the search area.

1 Introduction

Let $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_k\}$ be a finite alphabet and let $\mathcal{A} = \{A_i\}_{i=1}^m$ be a finite set of strings in Σ. The Longest Common Subsequence problem (LCS) is to find a longest sequence which is a common subsequence for all A_i $(i = 1, \ldots, m)$. The standard dynamic programming algorithm (DP) ([12]) solves the problem in $O(n^m)$ time, where n is the length of the longest sequence in the input. However, for many applications, such a running time is impractical, even if m is small (see [10]). This prompted a search for faster algorithms, possibly approximate, but with good performance bounds (see [2], [4], [8]).

In this paper, we present a learning algorithm which outputs an algorithm for LCS. For a given class of inputs, the algorithm builds a description of a *search area* based on the solutions produced by the dynamic programming algorithm. With every new training example, the modified search area—a subset of the dynamic programming matrix—is tested to determine the accuracy of the procedure for LCS defined by the search area. After accumulating enough training samples to reach a target approximation ratio, the learning algorithm returns an approximate procedure for solving LCS. Learning can be resumed to increase the approximation ratio of the procedure.

In our experiments, given a class of input strings, the learning algorithm is trained to construct a common subsequence whose length is at least 0.95[1] the length of the maximal common subsequence. We experimented with a variety of input classes, for which the strings are randomly generated according to probability distributions described by linear functions with different parameters.

We experimentally investigate the learning curve of the algorithm and the performance of the LCS-procedure after learning has been completed. In particular, the experiments with the class of two randomly generated 0,1-sequences of length n suggest that applying $O(n^{0.630})$ training samples guarantees an LCS-algorithm whose approximation ratio is 0.95 and the running time is $O(n^{1.654})$ (the standard algorithm for this case is quadratic). Other experiments show the performance of the algorithm on inputs with multiple sequences, alphabets with more than two letters, and different probability distributions of the letters in the input sequences. We compare the performance of our LCS-algorithm with the algorithm `Expansion` described in [2].

Note that our learning algorithm producing an LCS-procedure can also be viewed as an LCS-algorithm which improves its performance by alternating learning and testing.

Further on in the paper, the following terminology is used. For a sequence $A = A[i]_{i=1}^p$, $A[1..k]$ denotes the subsequence of A comprised of the first k entries of A. If $\mathcal{A} = \{A_k\}_{k=1}^m$ is a collection of sequences in an alphabet Σ, then $\texttt{length}[i_1, \ldots, i_m]$ denotes the length of the longest common subsequence of the sequences $A[1..i_1], \ldots, A[1..i_m]$.

[1] The constant 0.95 was arbitrarily selected for presentation; experiments were also conducted with approximation ratios ranging from 0.75 to 0.99.

2 Algorithm

Throughout this section, we describe the 2-dimensional case of LCS (the input is comprised of two strings); it is easy to see that the generalization to an arbitrary case is straightforward. Let A and B be two sequences of length p and q respectively. Recall that the standard quadratic dynamic programming algorithm for LCS, further called `DP`, computes `length`(p, q) by computing `length`$[i, j]$ for all $i \in [0, p]$, $j \in [0, q]$ for which $i + j < p + q$. To compute the longest subsequence itself, the algorithm backtracks its steps that led to computing the maximal length. This yields a sequence of pairs that trace computing `length`$[p, q]$. The sequence is denoted `Trace`(A, B); the $p \times q$-matrix is denoted `Length`(A, B).

```
Procedure DP(A, B)                          /* a modified DP is called Approximate-DP */
 1.  for (i = 1; i < p; i++)
 2.        length[i, 0] = 0;
 3.  for (j = 1; j < q; j++)
 4.        length[0, j] = 0;
 5.  for (i = 1; i < p; i++)
 6.       for (j = 1; j < q; j++)
 7.            { if (A[i] == B[j]) then
 8.                  length[i, j] = 1+ length[i − 1, j − 1];
 9.              length[i, j] = max ( length[i, j], length [i − 1, j],length[i, j − 1] )  };
   /* Computing length[p, q] has been completed */
10.  i = p; j = q; index = length[p, q]; Trace(index) = (p, q);
11.  while ( (i > 0)&(j > 0) ){
12.       if (A[i] == B[j])
13.            {  i − −; j − −; index = length[p, q] − 1;
14.               Trace(index) = (i, j); }
15.       else {
16.             if (length[i − 1, j] ≥ length[i, j])
17.                i − −;
18.             else
19.                j − −;
20.             index = length[i, j]; Trace(index) = (i, j); } }
21.  if ( (j == 0) & (i > 0) )
22.       for (t = i;  t ≥ 0; t − −){
23.            index−−; Trace(index) = (t, 0);}
24.  if ( (i == 0) & (j > 0) )
25.       for (t = j; t ≥ 0; t − −) {
26.            index−−; Trace(index) = (0, t);}
27.  return(Trace); return(index);
   /* Computing a longest common subsequence has been completed */
```

Figure 1: Standard dynamic programming algorithm for computing LCS

The idea of the learning algorithm is to restrict the computation of all entries of `Length`(A, B) to that of a subset which is essential to a given class of inputs. The construction of this subset, which we call the *search area* of the class, is done by accumulating traces of the solutions to sample inputs from the class. Once the search area is learned, the dynamic programming algorithm guided by the search area is run to compute a common subsequence.

It is convenient to associate `Length`(A, B) with a graph; every entry (i, j) of the matrix is represented by a vertex (i, j) of the graph $G_{p,q}$.

Definition. Let $G_{p,q}$ be a graph defined on the set of all pairs (i, j) $(i \in [0, p];\ j \in [0, q])$, where two distinct pairs (i, j) and (i', j') are adjacent, if $|i - i'| \leq 1$ and $|j - j'| \leq 1$. For every vertex $v = (i, j)$, i and j are called its coordinates. If $v = (i, j)$, $v' = (i', j')$ are two distinct vertices, and $i \leq i',\ j \leq j'$, then v is called smaller than v', written $v \prec v'$.

A simple path $P = \{v_i\}_{i=0}^{t}$ is called *monotone*, if for every $i = 0, \ldots, t-1$, v_i is adjacent to v_{i+1} and $v_i \prec v_{i+1}$.

A subset $SA \subseteq V$ is called a *search area* if $(0, 0), (p, q) \in SA$ and for every $v \in SA$, there is a monotone path in SA containing $(0, 0), v$, and (p, q). ∎

It is easy to see that the double loop in `DP` from Figure 1, can be generalized for an arbitrary search area SA; the only modification needed is for the case of a vertex $v \in SA$ which has neighbors that are smaller than v and are not in SA. Specifically, if $v[i-1, j-1]$ is missing, then lines 7 and 8 are not executed; if one or two of $(i-1, j), (i, j-1)$ are missing, the argument of function max (line 9) is respectively reduced. Finally, it can be readily seen that the second part of `DP`, from line 10 till line 26, do not depend on `SA`. This modification of `DP`, run according to a search area, is called `Approximate-DP( )`.

The other main components of the learning algorithm are: procedure `Generator`, which constructs inputs from a given class; procedure `Learn-area`, which constructs a search area using `Generator` and `DP`; and procedure `Test-area`, which runs `Approximate-DP` to test the accuracy of the current search area.

```
Procedure Learn-area( )
1.  for (i = 1; i ≤ training_number; i++)
2.        { Generate m sequences {A_k} using Generator;
3.          apply DP to compute LCS(A_1, ..., A_m);
4.          compute T = TRACE(LCS(A_1, ..., A_m));
5.          search-area = search-area ∪ T };
6.  Return search-area
```

Learning is done untill the *accuracy* of the algorithm achieves a preselected level.

```
Procedure Test-area(search-area)
1.  for (i = 1; i ≤ test_number; i++)
2.        { generate a sample using Generator;
3.          apply DP to compute max-length;
4.          apply Approximate-DP to compute approximate-length);
5.          ratio = approximate-length/ max-length;
6.          Ratio-sum = Ratio-sum + Ratio; }
7.  Accuracy = Ratio-sum/test_number;
```

The procedure `main()` below describes the algorithm that we used for the experiments.

```
main()
1.  Select Generator;
2.  while (Accurracy < Threshold)
3.        { Learn-area;
4.          Test-area;}
```

3 Experiments

The learning algorithm was tested using three sets of experiments. For every experiment, learning was continued until the approximation ratio of 0.95 was reached. For each training sample used to build the *search area*, 50 testing samples were used to determine the current approximation ratio.

The first set of experiments was to establish the learning curve and run-time performance of `Approximate-DP` for two equal-sized 0, 1-input strings. In the second set of experiments, we compared the performance of our algorithm with the algorithm `Expansion` described in [2]. In the third set of experiments, the learning algorithm was tested on classes of inputs with three strings; up to six letter alphabets; and strings generated with non-uniform distributions.

An input class is described by the number of input strings m, the size of the equal-sized input strings n, the number of symbols in the alphabet k, and the generation method (*Uniform* or *Non-uniform*). For *Uniform* generation, the probability of each alphabet symbol remains constant for the length of the string. The probabilities of the symbols are described by the set $p = (p_1, p_2, \ldots, p_k)$ where p_i refers to the probability of symbol i. For *Non-uniform*, the probability of each symbol is described by a linear function.

For all Tables, *Size* is the length of the equal-sized input strings; *Trials* is the number of training samples needed to reach the approximation ratio; *Speedup* is the ratio ($n \times n$/`search-area`); *Accuracy* is the approximation ratio reached through learning, and *Computations* is the size of the *search area*, e.g. the number of cells computed by `Approximate-DP`.

Learning Curve and Run-time Performance: The first set of experiments tested the performance of the algorithm for two equal-sized 0,1-input strings generated from two alphabet symbols. The input strings were generated using a uniform probability distribution with $p = (0.5, 0.5)$. The length n of the input strings was increased for each experiment. Thirty experiments were conducted and data points for $n = 100, 200, \ldots, 3000$ were obtained.

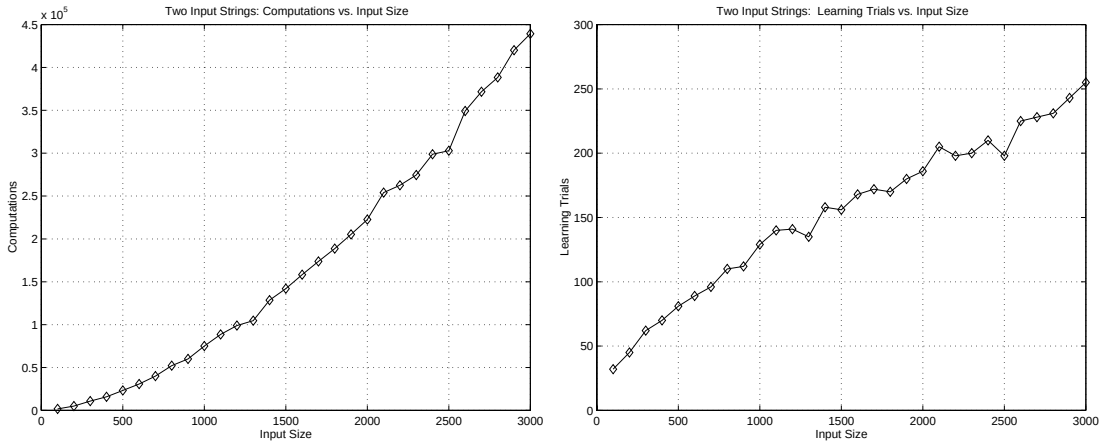

Figure 2: Runtime Measure and Learning Curve (Two Input Strings)
`InputClass` = *Uniform*, $m = 2, n = 100, \ldots 3000, k = 2, p = (0.5, 0.5)$

Size	*Trials*	*Speedup*	*Accurracy*	*Computations*
300	62	8.36	0.9512	10767
600	89	11.70	0.9536	30790
900	112	13.51	0.9504	59970
1200	141	14.57	0.9524	98996
1500	156	15.86	0.9504	142032
1800	170	17.18	0.9518	188625
2100	205	17.36	0.9531	254042
2400	210	19.31	0.9515	298735
2700	228	19.62	0.9506	371686
3000	255	20.49	0.9525	439258

Table 1: Performance Data (Two Input Strings)
`InputClass` = *Uniform*, $m = 2, n = 100, \ldots 3000, k = 2, p = (0.5, 0.5)$

The left (resp. right) plot in Figure 2 shows *Computations* (resp. *Trials*) as a function of the input size n. Using the least square method, we determined that among functions $a\ n^{\alpha} + b$, the data values for the runtime fits best the curve $0.903n^{1.654} + 207.5$ and the data for learning fits $1.563n^{0.630} + 1.181$. Recall that the running time of `DP` for this case is $O(n^2)$. Table 1 shows a sample of the results for the 30 experiments.

Learning Algorithm vs. Expansion Algorithm: The second set of experiments compares the learning algorithm with the `Expansion` algorithm for a class of two equal-sized 0, 1-input strings and a class of four equal-sized 0, 1-input strings. As before, the learning algorithm was trained to an approximation ratio of 0.95. For each experiment, 50 testing samples were used to test the accuracy of the approximate solution generated by learning algorithm and the `Expansion` algorithm. The accuracy was determined by the sum of the size of the approximate LCS divided by the optimal LCS for each sample divided by the number of testing samples. The average runtime (measured in *milliseconds*) of the 50 testing samples was computed.

Size Length	*Accurracy (Learn)*	*Accurracy (Expand)*	*Runtime (Learn)*	*Runtime (Expand)*	*Runtime (DP)*
200	0.9824	0.8780	13	2846	55
400	0.9757	0.8611	21	26493	342
600	0.9785	0.8585	95	87122	718
800	0.9769	0.8496	89	204412	1359
1000	0.9759	0.8460	198	432459	2342
1200	0.9777	0.8372	192	622376	2749
1400	0.9770	0.8422	216	984618	3724
1600	0.9765	0.8428	345	1508670	4670
1800	0.9748	0.8420	345	2203361	6356
2000	0.9733	0.8391	530	3182791	7769

Table 2: Learning vs. Expansion (Two Input Strings)
`InputClass` = *Uniform*, $m = 2, n = 100, \ldots, 2000, k = 2, p = (0.5, 0.5)$

Input Size	*Accurracy (Learn)*	*Accurracy (Expand)*	*Speedup (Learn)*	*Speedup (Expand)*	*Runtime (Learn)*	*Runtime (Expand)*	*Runtime (DP)*
10	0.9777	0.9108	9.59	959.00	20.0	0.2	191.8
20	0.9834	0.9315	23.26	1128.00	97.0	2.0	2256.0
30	0.9787	0.9183	32.46	2243.62	368.7	5.3	11966.0
40	0.9846	0.9191	49.77	2096.71	800.5	19.0	39838.5
50	0.9508	0.8862	54.17	2553.72	1697.0	36.0	91934.0

Table 3: Learning vs. Expansion (Four Input Strings)
`InputClass` = *Uniform*, $m = 4, n = 10, \ldots, 50, k = 2, p = (0.5, 0.5)$

For all 20 experiments using two inputs, the average LCS generate by the learning algorithm was larger than the average LCS generated by the `Expansion` algorithm. Since the run-time of the `Expansion` algorithm is $O(mn^2 \log n)$ [2], the learning algorithm, which speeds up the algorithm with respect to n, runs significantly faster for larger input. Table 2 shows a sample of the 20 experiments.

For the 5 experiments using four inputs, the learning algorithm generated a larger average LCS than the `Expansion` algorithm. But the running time of `Expansion` (linear on the respect on the number of input strings) runs significantly faster than the learning algorithm. Table 3 shows the experiments for four-input experiments.

Performance for Different Input Classes: The third set of experiments reveal the performance of the algorithm various classes of input. Figure 3 and Table 4 show the results of the experiments for three input strings. Note that the number of computations for $m = 3, n = 300$ is less than the number of computations for $m = 2, n = 3000$ from Table 1 even though the computation space of 300^3 is three times larger.

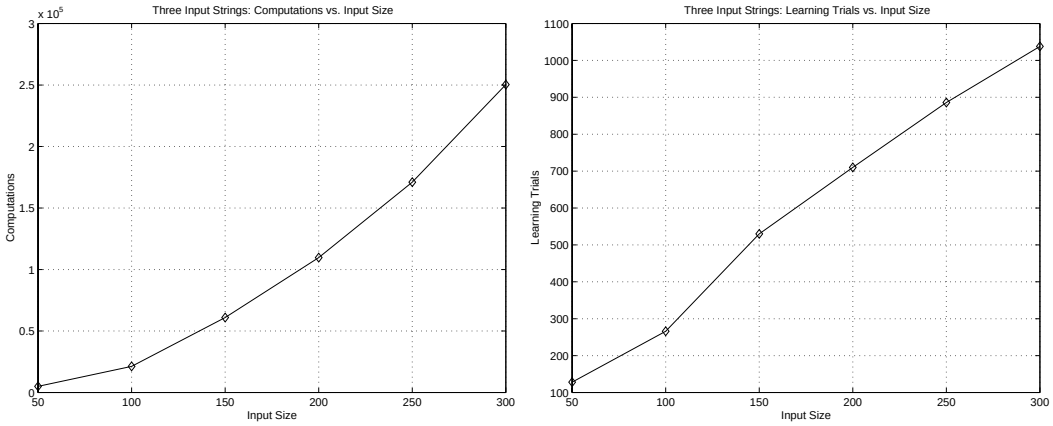

Figure 3: Runtime Measure and Learning Curve (Three Input Strings)
`InputClass` = *Uniform*, $m = 3, n = 50, \ldots 300, k = 2, p = (0.5, 0.5)$

Size	*Trials*	*Speedup*	*Accurracy*	*Computations*
50	128	25.08	0.9700	4984
100	266	47.03	0.9501	21264
150	530	55.34	0.9502	60987
200	710	72.93	0.9516	109694
250	886	91.35	0.9505	171045
300	1038	107.85	0.9500	250352

Table 4: Performance Data (Three Input Strings)
`InputClass` = *Uniform*, $m = 3, n = 50, \ldots, 300, k = 2, p = (0.5, 0.5)$

Table 5 shows the performance data for input strings generated from more symbols than two symbols. The number of computations increase as the number of symbols increases. This data indicates that there may be a relationship between the *search area* and the size of the LCS. Our testing revealed that increasing the number of symbols reduces the average size of the LCS for randomly generated input strings.

Symbols	*Trials*	*Speedup*	*Accurracy*	*Computations*
2	120	13.98	0.9518	71522
3	146	11.42	0.9507	87583
4	180	9.31	0.9511	107357
5	208	8.17	0.9509	122402
6	218	7.69	0.9522	129998

Table 5: Multiple Alphabet Symbols
`InputClass` = *Uniform*, $m = 2, n = 1000, k = 2, \ldots, 6, p = (p_1, \ldots, p_k)$

Table 6 shows the performance data for input strings generated from 0, 1-input strings with different probability ratios. The probability of the first symbol is equal to *ratio* and the probability of the

second symbol is equal to $1 - ratio$. The number of computations decreases as the symbol ratio becomes more disproportionate. Since the length of the LCS increases as the symbol become more disproportionate, this data also supports the notion that the size of the *search area* is proportional to the size of the LCS.

Ratio	*Trials*	*Speedup*	*Accurracy*	*Computations*
0.1	34	48.96	0.9506	20425
0.2	58	28.52	0.9524	35069
0.3	94	18.26	0.9522	54769
0.4	112	14.93	0.9506	66990

Table 6: Different Symbol Ratios
`InputClass` = *Uniform*, $m = 2, n = 1000, k = 2, p = (Ratio, 1 - Ratio)$

Table 7 shows selected experiments with input generated from non-uniform probability distributions The probability distribution of each 0, 1-input string is described by a linear functions defined over its length. For simplification, this function is defined by the line connecting the probability values of the leftmost string symbol p_l and the rightmost string symbol p_r. Therefore, a distribution defined by $p_l = 1$ and $p_r = 0$ describes a string with symbol 0 clustered heavily at the leftmost position, symbol 1 clustered at the rightmost position, and an even mix of both symbols at the center position. For a given set of inputs, the probability distributions are represented as a set of pairs $\{(p_{r_1}, p_{l_1}), (p_{r_2}, p_{l_2}), \ldots, (p_{r_m}, p_{l_m})\}$ Figure 4 shows the *search area* of the $n \times n$ computation space of the four experiments [2]. The shapes show the collection of solution paths that build the *search area*.

	$(p_l, p_r), (p_l, p_r)$	*Trials*	*Speedup*	*Accurracy*	*Computations*
1	$(0.5, 0.5), (0.5, 0.5)$	140	13.04	0.9528	76675
2	$(1.0, 0.0), (0.0, 1.0)$	215	11.33	0.9608	88256
3	$(0.6, 0.0), (0.0, 0.6)$	185	9.59	0.9505	104281
4	$(0.5, 1.0), (0.5, 0.0)$	140	18.79	0.9684	53214

Table 7: Non-uniform Input
`InputClass` = *Non-uniform*, $m = 2, n = 1000, k = 2$

Figure 4: *from left to right* Non-uniform Input Experiment 1, 2, 3, and 4

[2] These four experiments were chosen from a set of 40 experiments conducted on various input classes generated from non-uniform distributions.

4 Conclusions.

It is expected that a procedure which tunes an algorithm to a class of inputs should yield an algorithm which is significantly more efficient than a general algorithm for the problem. Our experiments support this expectation.

One of the most interesting results of our experiments with the learning algorithm is the discovery of the dependence of the search area on the probability distribution of the symbols in the input strings. In particular, the experiments suggest that the size of the search area is a small portion of the size of the dynamic programming matrix; for a problem with two input strings, the experiments indicate $o(n^2)$, where n is the length of the strings. It would be interesting to prove this experimental observation, as well as to prove that the number of samples needed to learn according to the target approximation ratio is sub-linear on n (equivalent to being logarithmic as a function of the size of the input space).

Unlike `Expansion`, `Approximate-DP` can be trained for any type of LCS; in particular, inputs of an arbitrary alphabet. In applications, our learning algorithm can be useful if the need to solve the problem occurs often and the inputs are of "the same" type. It appears that only a small number of initial inputs is needed to build up the search area to guide an efficient algorithm with an approximate ratio close to 1.

Our results suggest several interesting problems that can be approached experimentally. Clearly, the edit-problem ([12]) is very similar to LCS and a similar learning algorithm can be designed and experimentally tested. It is interesting to investigate how effective the usage of our strategy is for other applications of the dynamic programming paradigm. Finally, it is interesting and useful to resolve the problem of extrapolation of the results from small to very large input sizes, for which training is practically impossible.

References

[1] A. V. Aho, J. Hopcroft, and J. D. Ulman, *Data Structure and Algorithms*, Addison-Wesley, Reading , MA, 1983.

[2] P. Bonizzoni, M. D'Alessandro, G. Della Vedova, and G. Mauri, in Proceedings of "Algorithms and Experiments" (ALEX98), Trento, Italy, Feb 9-11, 1998, R. Battiti and A.A. Bertossi (Eds.) pp. 96 -102.

[3] P. Bonizzoni, and A. Ehrenfeucht, Approximation of the shortest common supersequence with ratio less than the alphabet size. Technical Report TR 149-95, Dipartimento di Scienze Della Informazione, Universitá Degli Studi di Milano, 1996.

[4] H. Buhrman, T. Jiang, M. Li, P. Vitányi, New Applications of the Incompressibility Method, (*to appear*).

[5] M.R. Garey, D.S. Johnson, *Computer and Intractability– A Guide to the Theory of NP-Completeness*, W.H. Freeman and Co., 1979.

[6] T. Jiang and M. Li, On the approximation of shortest common supersequences and longest subsequences, *SIAM J. on Computing,* 24(5), pp. 1122-1139, 1995.

[7] D. Maier, The complexity of some problems on subsequences and supersequences, *Journal of the ACM,* 25, pp. 322-336, 1978.

[8] M. S. Paterson and V. Dancik, Longest common subsequences, in Igor Prívara, Branislav Rovan and Peter Ruzicka, editors, *Mathematical Foundations of Computer Science,* 19^{th} *International Symposium*, vol. 841 of *LNCS*, pp. 127-142, Kosice, Slovakia, 1994.

[9] D. Sanfoff and J. B. Kruskal, *Time Wraps, SString Edits, and Macromolecules: The Theory and Practice of Sequence Comparions,* Addison-Wesley, Reading, MA, 1983.

[10] T. F. Smith and M. S. Waterman, Identification of common molecular subsequences, *Journal of Molecular Biology,* 147, pp. 195-197, 1981.

[11] L. G. Valiant, A theory of the learnable, *Communications of the ACM,* 27(11):1134-1142, 1984.

[12] R. A. Wagner and M. J. Fisher, The string-to-string correction problem, *Journal of the ACM* 21, pp. 168-173, 1974.