

Experimenting an Approximation Algorithm for the LCS^{*}

Paola Bonizzoni^{a,1}, Gianluca Della Vedova^{a,2},
Giancarlo Mauri^{a,3}

^a*Dipartimento di Informatica, Sistemistica e Comunicazione, Università degli Studi di Milano - Bicocca, via Bicocca degli Arcimboldi 8, Milano, I-20126, Italy*

Abstract

The problem of finding the longest common subsequence (lcs) of a given set of sequences over an alphabet Σ occurs in many interesting contexts, such as data compression and molecular biology, in order to measure the “similarity degree” among biological sequences. Since the problem is NP-complete in its decision version (i.e. does there exist a lcs of length at least k , for a given k ?) even over fixed alphabet, polynomial algorithms which give approximate solutions have been proposed. Among them, Long Run (LR) is the only one with guaranteed constant performance ratio.

In this paper, we give a new approximation algorithm for the longest common subsequence problem: the Expansion algorithm (EA). First of all we prove that the solution found by the Expansion algorithm is always at least as good as the one found by LR. Then we report the results of an experimentation with two different groups of instances, which show that EA clearly outperforms Long Run in practice.

1 Introduction

The problem of the longest common subsequence (LCS) is a well-known NP-hard problem [11]. Given two finite sequences $s = s_1s_2 \cdots s_m$, $t = t_1t_2 \cdots t_n$ over a finite alphabet Σ , then s is a *subsequence* of t if s can be obtained from t by removing some (possibly zero) symbols: for instance **five** is a subsequence

^{*} Research supported under MURST grant Cofinanziato *Bioinformatica e Scienze Biologiche*

¹ e-mail: bonizzoni@disco.unimib.it

² e-mail: dellavedova@disco.unimib.it

³ e-mail: mauri@disco.unimib.it

of reflexive. An instance of LCS is a set \mathcal{S} of sequences over Σ , and the solution is a longest sequence l over Σ such that l is a subsequence of each sequence in \mathcal{S} , that is a *longest common subsequence*, shortly *lcs*, for \mathcal{S} . The *lcs* of a pair of sequences is mainly related to the notion of edit distance, i.e. the number of editing steps (insertion or deletion of a single letter) required to obtain one string from the other one, but there are many others practical situations in which the LCS problem naturally arises. While traditional applications of it are in data compression, in syntactic pattern recognition and in file comparison [1] (for instance it is used in the *diff* command), recently the interest for finding efficient algorithms for it is mainly in the framework of molecular biology (the *lcs* is commonly used as a measure of similarity in the analysis of biological sequences [17]).

The problem of computing the *lcs* of two sequences has been deeply investigated (see the survey in [13]), and a number of algorithms have been proposed in order to improve the running time for typical instances [2,14,9,15], but all these algorithms still have a $O(n^2)$ time complexity in the worst case. The only algorithm that has broken this barrier is the one by Masek and Paterson [12] based on the Four Russians' technique [3]; their algorithm has $O(n^2/\log n)$ time complexity.

Let us now consider the more general problem of computing the *lcs* of k sequences of length n ; even this problem has been studied by several authors, proposing algorithms based on the dynamic programming technique [8,6], but they were not able to substantially improve the $O(n^k)$ time and space cost in the worst case. These requirements however are unacceptable even for small k in most situations, since the parameter n is usually extremely large in practice (e.g. text-editing, analysis of biological sequences, where n can be greater than 500). Consequently people have moved their interest towards the search for heuristic algorithms to find an approximate solution for the LCS problem. But, also in this framework, negative results have been provided for the LCS problem over an arbitrary alphabet: Jiang and Li [?] proved that the problem has no polynomial time approximation algorithm with performance ratio n^δ , for any constant $\delta < 1$, unless $P = NP$. Despite the discouraging results, it has been proved that the LCS problem over a fixed alphabet can be indeed very well approximated on the average by using a simple algorithm called Long Run [?]: it gives as a solution of the LCS for a set S of sequences, the sequence σ^l , such that σ^l is the longest common subsequence of S of the form σ^l , for $\sigma \in \Sigma$. The Long Run algorithm works quite well in practice since it can provide a solution which has a length close to the optimum. More precisely, in [?], Jiang and Li proved that given n input sequences generated randomly according to the uniform probability distribution, all of the same length n and over fixed alphabet, then the Long Run algorithm approximates the *lcs* with an $O(n^{1/2+\epsilon})$ expected additive error (the additive error is given as the difference between the length of the optimum and of the approximation)

for any $\epsilon > 0$. Anyway this does not imply that Long Run performs well on instances made of a relatively small number (e.g. less than 30) of long sequences (more than 100 characters each). Moreover, even though Long Run gives a solution whose length is a good approximation of the optimal one, it is quite evident that this algorithm presents some deep shortcomings that are more relevant in the molecular biology setting. In fact the LCS is used in comparing biological sequences [5] and the actual lcs of a set of biological sequences should represent some regions that are common to all sequences, hence are highly conserved regions. This means that an lcs is a good candidate to contain encoding regions, that is subsequences of high biological relevance. Anyway it is not hard to realize that the subsequence given by Long Run seldom has a biological meaning, as it contains only one distinct symbol.

Another drawback of using Long Run to compute a subsequence is that it does not give the optimal solution even when the instance contains only one sequence, and hence the optimal solution is trivially the sequence in the instance. Moreover it is possible to prove that there are instances consisting of one sequence where Long Run gives a subsequence whose length is $1/|\Sigma|$ of that of the actual lcs, where $|\Sigma|$ is size of the alphabet Σ . Actually, it is not difficult to prove that $|\Sigma|$ is also the guaranteed performance ratio of Long Run.

In this paper we give a new approximation algorithm, called *Expansion*, for the LCS problem, which has a guaranteed performance ratio $|\Sigma|$, hence matching the one of Long Run, even though such bound is not tight for our algorithm. Moreover we will show experimentally that the average performance of our algorithm is definitely better than the one of Long Run. The algorithm is based on a technique similar to the one initially proposed for the Shortest Common Supersequence problem in [?]. The experiments have been executed on two main groups of instances: one consisting of sequences of length between 90 and 100 and the other consisting of sequences of length between 400 and 500.

The goals of the two experiments are different. In the first one we have instances containing random sequences and we compare the approximate solution with the optimum one, so that we can compute exactly the performance ratio of the algorithm on these instances. The instances of the second experiment contain a greater number of longer sequences, moreover the sequences are fairly homologous in order to point out the behavior of the algorithm over biological sequences. In fact each instance contains sequences generated from a random sequence simulating an evolution according to the Jukes-Cantor [10] model of evolution, hence are sufficiently representative of the sequences usually found in practice. Instances generated in this way contain sequences of at least 400 symbols, thus ruling out the possibility of comparing the approximate solution with the exact solution, since a dynamic programming algorithm for

such instances would not be feasible. Nonetheless we are able to provide an upper bound on the performance ratio of the algorithm, based on the fact that a common subsequence of a set S of sequences cannot be longer than the shortest sequence in S . Since such bound is trivial we expect that the performance ratio of our algorithm is definitely better than the one we have obtained.

A comparison between the results obtained from the two main experiments has allowed to confirm that the Expansion algorithm achieves an average performance ratio less than 1.08, while the Long Run has an average performance ratio which is at least 1.34.

2 Preliminaries

Let Σ be a finite alphabet. As usual we will denote by Σ^* the set of sequences of symbols from Σ . For a given $\sigma \in \Sigma$ and $i > 0$, i integer, σ^i will denote the sequence of length i containing only the symbol σ . The length of a sequence $s \in \Sigma^*$, that is the number of symbols that are in s , is denoted by $|s|$. A *basic sequence* of length k over Σ is a sequence $\sigma_1 \cdots \sigma_k$, with $\sigma_i \in \Sigma$ for all $1 \leq i \leq k$ and $\sigma_i \neq \sigma_{i+1}$ for all i , $1 \leq i < k$; note that there are $|\Sigma|(|\Sigma| - 1)^{k-1}$ basic sequences of length k over an alphabet Σ . A *stream* of a set \mathcal{S} of sequences is a basic sequence that is a common subsequence of \mathcal{S} . When \mathcal{S} contains only a sequence s , by stream of s we mean the stream of $\{s\}$. Given a sequence s its *factorization* into blocks is the sequence $(\sigma_1, j_1), \dots, (\sigma_k, j_k)$, where $\sigma_1 \cdots \sigma_k$ is a stream of s , $j_l > 0$, for $1 \leq l \leq k$ and $\sigma_1^{j_1} \cdots \sigma_k^{j_k} = s$. The sequence $\sigma_i^{j_i}$ is called the i -th *block* of s . The *run* of a sequence is the maximum length of one of its blocks.

The following set of instances will be used as an example. Let \mathcal{S} be the set $\{aabbaabcb, abbbcbabbba, bcabbab\}$, then $\{(a, 2)(b, 2)(a, 2)(b, 1)(c, 1)(b, 1)(c, 1), (a, 1)(b, 4)(c, 1)(b, 1)(a, 1)(b, 2)(a, 2), (b, 1)(c, 1)(a, 1)(b, 3)(a, 1)(b, 1)\}$ is the set of the factorizations of each sequence in \mathcal{S} . The basic sequences of length 2 over the alphabet $\{a, b, c\}$ are ab, ac, ba, bc, ca, cb , moreover ab is a stream of \mathcal{S} while ca is not. Note that the sequences in \mathcal{S} have run 2, 4, 3 respectively.

Let $c_A(\mathcal{S})$ be the common subsequence which is returned as a solution by an approximation algorithm A for the instance \mathcal{S} of the LCS problem, and let $opt(\mathcal{S})$ be the optimum, i.e. the length of a lcs for \mathcal{S} . Then, the *performance ratio* of the algorithm over an instance \mathcal{S} is the value $R_A(\mathcal{S})$, where

$$R_A(\mathcal{S}) = \frac{opt(\mathcal{S})}{|c_A(\mathcal{S})|}$$

Note that $R_a(\mathcal{S})$ is always bigger than or equal to 1, and that the closer it is

to one, the better the approximate solution is. We say that algorithm A has the *guaranteed performance ratio* r , if $R_A(\mathcal{S}) \leq r$, for every instance \mathcal{S} .

There exists a dynamic programming algorithm to compute the lcs of a set \mathcal{S} of k sequences [16], but it requires $O(n^k)$ time and space, where n is the length of the longest sequence in \mathcal{S} . Hence this algorithm is feasible only for small values of n and k . An improvement of such algorithm based on the Four Russians' technique [3] is possible and the time complexity would be $O(n^k / \log n)$, but the hidden constants would not make such an algorithm more appealing than the one in [16] for practical cases. It is possible to generalize Hirschberg's algorithm [7] for the LCS of 2 sequences to our problem, leading to an $O(n^k)$ time and $O(n^{k-1})$ space algorithm, but with a time complexity that is twice as much as the one of the dynamic programming algorithm described in [16].

When the number k of sequences is not fixed, but it is a part of the instance, the time complexities of the algorithms before mentioned are not polynomial. In fact it is known from [11] that in such case the decision version of the LCS problem is *NP*-complete also over binary alphabet. Hence the subsequent step is to look for efficient approximation algorithms; an example of such an algorithm is the Long Run.

It is rather straightforward to describe the Long Run algorithm. Given a set \mathcal{S} of sequences over the alphabet Σ , let $\text{occur}_\sigma(s)$ be the number of occurrences of σ in a sequence s . Then, for each symbol $\sigma \in \Sigma$, let c_σ be the minimum value of $\text{occur}_\sigma(s)$ over all sequences $s \in \mathcal{S}$. Long Run returns α^{c_α} where α is the symbol of Σ maximizing c_α .

While the Long Run algorithm over fixed alphabet gives an approximate common subsequence with a guaranteed performance ratio of $|\Sigma|$, it is easy to note that such subsequence contains only one symbol of the alphabet Σ , so it is rather useless in practice.

3 The Expansion algorithm

In this section we propose a different approach for computing an approximate solution of the LCS problem. Our algorithm is based on the following main observation: an lcs of a set \mathcal{S} of sequences has a number n of blocks, with $1 \leq n \leq B$, for B the length of a stream for \mathcal{S} of minimum length. A strategy to obtain a good approximation consists of expanding streams of different lengths, so that we obtain a solution with factorization similar to the one of a lcs. Since the number of possible expansions of a stream (that is assigning an exponent to each symbol of the stream) is exponential in n , we develop a polynomial time strategy which takes into account the variation in the size

of the blocks inside the factorizations of the sequences in \mathcal{S} : this behavior is realized by the *Expand* procedure.

The *Expand* procedure receives as input a set \mathcal{S} of sequences and a stream e . It scans e from left to right, block by block, and at each time it tries to obtain a new common subsequence of \mathcal{S} by doubling the length of the examined block. Then, it continues to expand the sequence from left to right in this way, until no block of the sequence can be doubled. Finally, it examines again the sequence from left to right, trying each time to enlarge the size of each block, until the sequence cannot be further expanded, since otherwise the property of being a common subsequence of \mathcal{S} is violated.

Expand(\mathcal{S}, e)

Input: A set \mathcal{S} of sequences and a basic sequence $e = \sigma_1 \cdots \sigma_{|e|}$

Pose $k_i = 1$ for all $1 \leq i \leq |e|$

Repeat

test := false

For $1 \leq j \leq |e|$ do

If $\sigma_1^{k_1} \cdots \sigma_j^{2k_j} \cdots \sigma_{|e|}^{k_{|e|}}$ is a subsequence of S then

$k_j := 2 * k_j$

test := true

EndIf

EndFor

Until test = false

For $1 \leq j \leq |e|$ do

$k_j := \max\{\alpha\}$ such that $\sigma_1^{k_1} \cdots \sigma_j^\alpha \cdots \sigma_{|e|}^{k_{|e|}}$ is a common subsequence of S

EndFor

Return($\sigma_1^{k_1} \cdots \sigma_j^{k_j} \cdots \sigma_{|e|}^{k_{|e|}}$).

Computing the longest size of a block can be implemented with a binary search.

A simple example will help the reader in understanding the procedure. Let \mathcal{S} be the set $\{a^4b^3a^4b^2a, a^3b^4a^4b^3\}$, then the sequences of the expansions of the stream $abab$ computed inside *Expand* is: $abab, a^2bab, a^2b^2ab, a^2b^2a^2b, a^2b^2a^2b^2, a^2b^2a^4b^2, a^3b^2a^4b^2, a^3b^3a^4b^2$. The latter sequence is the one returned by the procedure.

The *Expand* procedure gives a way to obtain a common subsequence from a stream. The basic idea on which the *Expansion* algorithm relies is computing the lcs of a set \mathcal{S} of sequences from a set T of streams of \mathcal{S} , where each sequence x in T is expanded by the *Expand* procedure. At the end, a set C of common subsequences for \mathcal{S} is obtained: then the sequence of maximum

length in C is the solution returned by the algorithm.

Computing a set of streams requires two different approaches depending on the size of the alphabet; in fact the set of all streams over a binary alphabet is polynomial in the size of the instance, hence it is possible to compute all of them. This is not true in the case of arbitrary alphabets, where we have to use some sort of heuristic to compute a subset of all possible streams.

3.1 Binary alphabet

Let us first illustrate the main body of the algorithm in the case of binary alphabet: all streams of the instance are computed and then expanded by the **Expand** procedure. The best sequence among all returned by the various calls to **Expand** is the output of the algorithm.

ExpansionBinary(\mathcal{S})

Input: A set \mathcal{S} of sequences.

Let B be the minimum length of a stream of \mathcal{S} .

For $1 \leq t \leq B$ do

$z_t = \mathbf{Expand}(\mathcal{S}, \sigma_1 \sigma_2 \cdots \sigma_t)$, where $\sigma_1 = 0$ and $\sigma_1 \cdots \sigma_t$ is a stream of \mathcal{S}

$w_t = \mathbf{Expand}(\mathcal{S}, \sigma_1 \sigma_2 \cdots \sigma_t)$, where $\sigma_1 = 1$ and $\sigma_1 \cdots \sigma_t$ is a stream of \mathcal{S}

EndFor

Let cs be the longest sequence in the set $\{z_t : 1 \leq t \leq B\} \cup \{w_t : 1 \leq t \leq B\}$

Return(cs)

Note that given the instance $\mathcal{S} = \{a^4 b^3 a^4 b^2 a, a^3 b^4 a^4 b^3\}$ of the example in the previous section, the **Expansion** algorithm computes on input \mathcal{S} an approximate solution of length at least 12, while **Long Run** returns the subsequence a^7 .

3.2 Arbitrary alphabet

As stated previously, in the case of an arbitrary alphabet there is an additional difficulty w.r.t. to the binary case in applying the technique of the **Expand** procedure, which is given by the fact that the number of the streams of the instance may be exponential in the length of the sequences in the instance. Thus we need to develop an heuristic to choose a subset of streams that will be expanded by the **Expand** procedure. The heuristic we give consists of two steps. Given a set \mathcal{S} of sequences, we initially compute all streams of \mathcal{S} of maximum length 2. Then we apply a *greedy* algorithm to \mathcal{S} to obtain a stream st of \mathcal{S} . All substrings of st are streams of \mathcal{S} that are expanded by the **Expand**

procedure. The algorithm is stated below, where we assume that an exact lcs of two sequences is computed by the dynamic programming algorithm in [4].

Greedy(\mathcal{S})

Input: A set $\mathcal{S} = \{s_1, \dots, s_{|\mathcal{S}|}\}$ of sequences.

If \mathcal{S} contains only one sequence then

Return the sequence in \mathcal{S}

EndIf

If $|\mathcal{S}| = 2$ then

$lcs := \text{LCS}(s_1, s_2)$

Return the longest stream of lcs

EndIf

For each $s_i \in \mathcal{S}$ do

Replace s_i with the longest stream of $\{s_i\}$

EndFor

For each i, j such that $1 \leq i < j \leq |\mathcal{S}|$ do

$s_{i,j} = |\text{Greedy}(\{s_i, s_j\})|$

$\text{temp}[i, j] := |s_{i,j}|$

EndFor

Let i, j be the pair of indices that maximizes $\text{temp}[i, j]$

Return($\text{Greedy}(\mathcal{S} - \{s_i, s_j\} \cup s_{i,j})$)

ExpansionArbitrary(\mathcal{S})

Input: A set \mathcal{S} of sequences.

$\text{Streams} := \{s : s \text{ is a stream of } \mathcal{S} \text{ of length } \leq 2\}$

Add to Streams all substrings of $\text{Greedy}(\mathcal{S})$

Initially cs is the empty word;

For each $z \in \text{Streams}$ do

$w = \mathbf{Expand}(\mathcal{S}, z)$

If w is longer than cs then

$cs := w$

EndIf

EndFor

Return(cs)

4 Theoretical analysis

Given a set \mathcal{S} of n sequences of length m , testing if a sequence is a common subsequence of \mathcal{S} can be done in $O(nm)$ time. A careful implementation of

Expand requires exactly 1 unsuccessful test and at most $O(\log m)$ successful tests for each block of the stream e . Consequently the total time to compute the exponent of each block is $O(\log m)$, as the last step is a binary search. Since there are at most m blocks, the time complexity of Expand is $O(nm^2 \log m)$.

The ExpansionBinary algorithm contains at most $2m$ calls to Expand, consequently the algorithm has $O(nm^3 \log m)$ time complexity.

The analysis of ExpansionArbitrary is slightly more involved. A careful implementation of the Greedy procedure has $O(n^2m^2)$ time complexity when it receives as input the set \mathcal{S} of sequences, while it requires $O(m^2)$ when it receives 2 sequences of maximum length m as input. The substrings of the output of Greedy(\mathcal{S}) are at most m^2 , as such output must be a common subsequence of \mathcal{S} , consequently the streams that are expanded are at most $|\Sigma|^2 + m^2$. It follows that the time complexity of the algorithm is $O((|\Sigma|^2 + m^2)(nm^2 \log m) + n^2m^2) = O((|\Sigma|^2 + m^2) \log m + n)nm^2$, for $|\Sigma| > 2$. Consequently when $m > n$ and $m > |\Sigma|$, as in the instances of our experiments the time complexity is $O(nm^4 \log m)$.

Observe that we can describe the Long Run algorithm as a restricted case of the *Expansion* algorithm, thus showing that the solution computed by such algorithm over a set \mathcal{S} of sequences can never be better than the solution computed by our *Expansion* algorithm.

Long Run(\mathcal{S})

Input: A set \mathcal{S} of sequences.

Initially lcs is the empty word;

For each $z \in \Sigma$ do

$w = \mathbf{Expand}(\mathcal{S}, z)$

If w is longer than cs then

$cs := w$

EndIf EndFor

Return(cs)

Hence the following results are immediate, since the set of streams expanded by LR is a subset of those expanded by EA.

Theorem 4.1 *For every set S of sequences over Σ , given $c_{LR}(\mathcal{S})$ the approximate solution of the Long Run and $c_{EA}(\mathcal{S})$ the solution of the Expansion algorithm, then $|c_{EA}(\mathcal{S})| \geq |c_{LR}(\mathcal{S})|$.*

Corollary 1 *The Expansion algorithm has $|\Sigma|$ guaranteed performance ratio.*

5 Experimental analysis

In this section, we describe the results of two different groups of experiments we have developed to study the average case behavior of our algorithm. The first group contains instances with 4 random sequences of length between 90 and 100, where the runs of the sequences are generated according to the uniform distribution. For these experiments we have been able to compare the approximate solution computed by the Expand algorithm with the one returned by Long Run and an exact lcs obtained by the dynamic programming algorithm. Besides a natural comparison based on the lengths of the solutions, we propose a measure representing how much the approximate solution resembles an optimal one. To achieve this goal we introduce a new parameter, called *similarity*, which is defined by the following formula relating the number $N(\mathcal{S})$ of blocks of a lcs over the set \mathcal{S} of sequences to the number $A(\mathcal{S})$ of blocks of the approximate solution: $\sqrt{E((N(\mathcal{S}) - A(\mathcal{S}))^2)}$, where $E()$ is the expectation. Please note that it is desirable to have an algorithm which achieves a small similarity index.

The second group of experiments consists of instances with 5, 10 or 20 sequences whose lengths range from 400 to 500. Moreover the sequences in each set \mathcal{S} are generated from a random sequence $base(\mathcal{S})$ on which we simulated an evolution process according to the Jukes-Cantor model [10]. Moreover in our simulation only deletions and substitutions were allowed. In this way we can easily generate instances that are representative of the ones usually found in practice. It has not been possible to compute the exact solution of the LCS over such instances, due to both time and space constraints⁴, hence we have compared the length of our approximate solution with that of the shortest sequence in \mathcal{S} , which is a (trivial) upper bound on the length of a lcs. Consequently the ratios stated in Tables 2 are upper bounds of the actual ones. Since we did not compute the actual lcs, it did not make sense to compute the similarity index, hence in this case we dealt only with the performance ratio. A fundamental parameter in all experiments is the maximum run of the sequences in the instances.

The results of the first group of experiments are summarized in Table 1, where the the average performance ratio, the standard deviation of the performance ratio and the similarity index achieved by Expansion algorithm and Long Run are represented. The sequences of the experiments are over binary alphabet and are obtained by generating sequences of integer values according to a uniform distribution in the range between 1 and the maximum run: each

⁴ The space constraints are especially demanding, as a careful implementation of Hirschberg's algorithm for instances of 5 sequences of length 400 still requires at least 16Gbytes of memory

of such sequence gives the lengths of the blocks. In particular this group of experiments contains input sequences with length between 90 and 100.

Max Run		2	6	12	18
Expansion	Average R_A	1.0715	1.063	1.0496	1.0416
	Std. Dev.	0.0205	0.0299	0.0374	0.0412
	Similarity	6.2914	3.3651	2.2081	1.531
Long Run	Average R_A	1.6224	1.4176	1.3717	1.3456
	Std. Dev.	0.0428	0.0742	0.1083	0.1366
	Similarity	50.779	15.9346	7.3655	4.5989

Table 1

Experiments over sequences of length between 90 and 100

In Table 1 it is possible to note that the Expansion algorithm has outperformed the Long Run algorithm for each value of the maximum run parameter giving, on the average, a better solution both in terms of the length of the approximate solution and in terms of the number of blocks of the approximate solution w.r.t. the number of blocks of an actual lcs. In fact the Long Run algorithm has an average performance ratio which is always at least 1.34, while the Expansion algorithm has never achieved an average performance ratio greater than or equal to 1.08. The analysis of the similarity index shows clearly that the Expansion algorithm compute an approximate solution which is more similar to an actual lcs, as the similarity index of Long Run is always at least three times as the one of Expansion.

The second group of experiments have been run over sequences of maximum length 500 and alphabets of sizes 4 and 20, that is using the alphabet of DNA and protein sequences respectively. The results that we have obtained are very encouraging, since the Expansion algorithm has never had an average performance ratio larger than 1.16.

Studying how the performance of our algorithm depends on the size of the alphabet has been one of the goals of this paper. While it is obvious that the algorithm should perform better on alphabets of smaller size, we found out that the performance of the algorithm smoothly get worse when alphabet size increases from 4 to 20 (see Table 2).

Another goal of our experiments has been determining how the performance ratio is influenced by the number of sequences in each instance. In this case the degradation of the performance w.r.t. the size of the instances is noticeable, but it is still smooth. The Fig. 1 reports only part of the results shown in Table 2, pointing out the dependence of the performance of the algorithm w.r.t. the minimum length of the sequences and the number of sequences in

Min. length	Max Run		
	8	16	32
400	1.15024	1.08671	1.05411
450	1.04483	1.0266	1.0171
480	1.00787	1.00409	1.0021

20 sequences, alphabet size 4

Min. length	Max Run		
	8	16	32
400	1.15664	1.08485	1.05187
450	1.04413	1.02766	1.01634
480	1.00757	1.00397	1.00209

20 sequences, alphabet size 20

Min. length	Max Run		
	8	16	32
400	1.13848	1.07975	1.04457
450	1.03437	1.0206	1.0117
480	1.00565	1.00333	1.00124

10 sequences, alphabet size 4

Min. length	Max Run		
	8	16	32
400	1.13308	1.08091	1.04814
450	1.03421	1.02097	1.01232
480	1.00508	1.00231	1.00129

10 sequences, alphabet size 20

Min. length	Max Run		
	8	16	32
400	1.11052	1.05749	1.03417
450	1.01994	1.01274	1.00692
480	1.00361	1.00154	1.00074

5 sequences, alphabet size 4

Min. length	Max Run		
	8	16	32
400	1.10212	1.05383	1.03377
450	1.02046	1.01149	1.00786
480	1.00271	1.05383	1.00067

5 sequences, alphabet size 20

Table 2

Results of the experiment over sequences of maximum length 500

the instance. Such results are from experiments over sequences with maximum run 16 and alphabet size 4 are represented. Anyway, the trends of the results we have obtained for different maximum runs and different alphabet sizes are similar to the ones reported in such figure.

The third goal of the experiments has been to determine how the performance ratio of the algorithm depends on the maximum run of the sequences. An interesting fact that it is possible to devise from the results of the experiments is that the performance of the algorithm improves as the maximum run increases. This may seem quite surprising, but the Expand procedure is designed so that it is able to adapt its behavior according to the distribution of the runs in the sequences. In Table 2, which summarizes the results of this group of experiments we have not stated the standard deviation, since the values found

Fig. 1. Experiments with maximum run 16
are along the same lines as those in the first experiment.

6 Conclusions

In the paper we have described the Expansion algorithm (EA), a new approximation algorithm for the longest common subsequence problem, and we have shown experimentally that it performs better than Long Run on the average. By running experiments on different alphabets, sequence runs, sequence lengths and instance sizes we have also proved the effectiveness of the algorithm for practical cases (i.e. biological sequences).

The Expansion algorithm, as pointed out in the paper, owes its simplicity and effectiveness to the Expand procedure, which implicitly takes into account the distribution of symbols of the specific instance. An interesting possible extension of our work might be to develop new expansion techniques that exploit a more refined analysis of the distribution of the symbols, such as randomized algorithms.

References

- [1] A. Aho, J. Hopcroft, and J. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, MA, 1983.
- [2] A. Apostolico and C. Guerra. The longest common subsequence problem revisited. *Algorithmica*, 18(1):1–11, 1987.
- [3] V. Arlazarov, E. Dinic, M. Kronrod, and I. Faradzev. On the economic construction of the transitive closure of a direct graph. *Soviet. Math. Dokl.*,

11(5):1209–1210, 1970.

- [4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.
- [5] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, Cambridge, 1997.
- [6] K. Hakata and H. Imai. The longest common subsequence problem for small alphabet size between many strings. In *Proc. 3rd International Symposium on Algorithms and Computation (ISAAC)*, volume 650 of *LNCS*, pages 469–478. Springer Verlag, 1992.
- [7] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of ACM*, 18(6):341–343, 1975.
- [8] W. Hsu and M. Du. New algorithms for the LCS problem. *Journal of Computer and Systems Sciences*, 19:133–152, 1984.
- [9] G. Jacobson and K.-P. Vo. Heaviest increasing/common subsequence problem. In *Proceedings of the 3rd Combinatorial Pattern Matching*, volume 644 of *LNCS*, pages 52–66, 1992.
- [10] W.-H. Li. *Molecular Evolution*. Sinauer Assoc., 1997.
- [11] D. Maier. The complexity of some problems on subsequences and supersequences. *Journal of the ACM*, 25:322–336, 1978.
- [12] W. Masek and M. Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*, 20(1):18–31, 1980.
- [13] M. Paterson and V. Dancik. Longest common subsequences. In *Mathematical Foundations of Computer Science, 19th International Symposium (MFCS)*, volume 841 of *LNCS*, pages 127–142, 1994.
- [14] P. Pevzner. Matrix longest common subsequence problem, duality and Hilbert bases. In *Proceedings of the 3rd Combinatorial Pattern Matching*, volume 644 of *LNCS*, pages 79–89, 1992.
- [15] C. Rick. A new flexible algorithm for the longest common subsequence problem. In *Proceedings of the 5rd Combinatorial Pattern Matching*, volume 937 of *LNCS*, pages 340–351, 1995.
- [16] D. Sankoff and J. Kruskal, editors. *Time Warps, String Edits and Macromolecules: the Theory and Practice of Sequence Comparisons*. Addison Wesley, 1983.
- [17] T. Smith and M. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.