Bounding the Expected Length of Longest Common Subsequences and Forests

Ricardo A. Baeza-Yates[1 2] Ricard Gavaldà[1] Gonzalo Navarro[2]

[1] Dept. LSI, Technical Univ. of Catalunya, Pau Gargallo 5, 08028 Barcelona, Spain
[2] Depto. de Cs. de la Computación, University of Chile, Casilla 2777, Santiago, Chile
[3] This work has been partially supported by the ESPRIT Long Term Research Project 20244, ALCOM IT. E-mail: {rbaeza,gavalda}@goliat.upc.es, gnavarro@dcc.uchile.cl.

Abstract. We present two techniques to find lower and upper bounds for the expected length of longest common subsequences and forests of two random sequences of the same length, over a fixed size, uniformly distributed alphabet. We emphasize the power of the methods used, which are Markov chains and Kolmogorov complexity. As a corollary, we obtain some new lower and upper bounds for the problems mentioned.

1 Introduction

The longest common subsequence (LCS) of two strings is one of the main problems in combinatorial pattern matching. The LCS problem is related to DNA or protein alignments, file comparison, speech recognition, etc. We say that x is a subsequence of u if we can obtain x by deleting zero or more characters of u. The LCS of two strings u and v of length n is defined as the longest subsequence x common to u and v. For example, the LCS of *longest* and *large* is *lge*. An open problem related to the LCS is its expected length for two random strings of length n over a uniformly distributed alphabet of size k, denoted by $EL_n^{(k)}$. In particular, if an alignment or common subsequence of two given sequences is relatively larger than $EL_n^{(k)}$, we may infer that it is more than a coincidence, and that the result should be studied further. If $\ell cs(u,v)$ denotes the length of the LCS for two strings u and v, we have:

$$EL_n^{(k)} = \frac{1}{k^{2n}} \sum_{|u|=|v|=n} \ell cs(u,v)$$

Because $EL_n^{(k)}$ is superadditive, that is, $EL_n^{(k)} + EL_m^{(k)} \le EL_{n+m}^{(k)}$, it is possible to show [CS75] that

$$\gamma_k = \lim_{n\to\infty} \frac{EL_n^{(k)}}{n} = \sup_n \frac{EL_n^{(k)}}{n}$$

exists. However, the exact values of γ_k are still not known. For that reason, several lower and upper bounds have been devised for γ_k. For example, it is known that

$$1 \le \gamma_k \sqrt{k} \le e$$

First we present new lower bounds for $k > 2$ for the LCS. These new results are based on a new class of automata (following the work of Deken [Dek79] and Dančík & Paterson [Dan94, PD94]) that simulates an algorithm that computes the LCS over two random infinite strings. These automata are called CSS (Common SubSequence) machines in [Dan94].

To obtain upper bounds, we refine and extend the Kolmogorov complexity approach mentioned by Li and Vitányi [LV93], which is simple and elegant. Kolmogorov complexity has been very useful in many areas of computer science. The reader is referred to the monograph of Li and Vitányi [LV93] for a very complete treatment of the origins, development, and applications of this concept.

We also apply both techniques to a generalization of the LCS problem, called the Longest Common Forest (LCF) by Pevzner and Waterman [PW93], obtaining the first known lower and upper bounds for the expected size of the LCF of two random sequences. In particular, we show that for large alphabets, the expected length of the LCF is also upper bounded by $e/\sqrt{k}$.

2 Longest Common Subsequences and Forests

The LCS of two strings u and v can be computed using dynamic programming over a matrix L defined by $L[0,i] = L[i,0] = 0$ for $0 \leq i \leq n$ and

$$L[i,j] = \max(L[i-1,j], L[i,j-1], L[i-1,j-1] + (u[i] =?v[j])), 1 \leq i,j \leq n$$

where $(u[i] =?v[j])$ is defined as 1 if both characters are equal, or 0 otherwise. The length of the LCS is given by $L[n,n]$. This algorithm can be implemented using only $2n^2$ comparisons. For faster algorithms which solve the LCS we refer the reader to [GBY91, PD94, Ric95].

Longest Common Forests (LCF) are defined in [PW93] as one particular case of general alignments between strings, called the A-LCS problem. Basically, in a LCF we allow a character to match more than one character of the other sequence, but if we look at every match as an edge between the two sequences, then no edge crossings can exist. Hence, the alignment is a set of trees or forest. In [PW93] a cn^2 algorithm to compute the A-LCS problem is given, where c is related to the determinant of a matrix defining the generalized alignment rules. They mention that $c = 2$ for the LCF problem, but a simple algorithm is not explicitly given. In fact, the dynamic programming procedure for LCF is given by

$$L[i,j] = \max(L[i-1,j], L[i,j-1]) + (u[i] =?v[j])$$

which requires only $2n^2$ comparisons. If $\ell cf(u,v)$ denotes the length of the LCF for two strings u and v, in general we have

$$0 \leq \ell cf(u,v) \leq 2(|u| + |v|) - 1$$

where the upper bound can be seen as the longest path where we either advance in a row or a column of the matrix L. Similarly to the LCS, LCF is superadditive.

We can define

$$EF_n^{(k)} = \frac{1}{k^{2n}} \sum_{|u|=|v|=n} \ell cf(u,v)$$

and

$$f_k = \lim_{n\to\infty} \frac{EF_n^{(k)}}{n} = \sup_n \frac{EF_n^{(k)}}{n} \le 2 \ .$$

Figure 1 shows some examples of LCFs as well as the corresponding LCS length (the solutions shown in the examples are not necessarily unique).

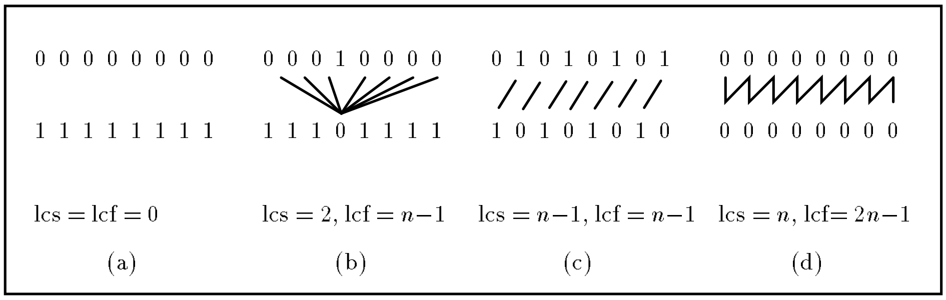

Fig. 1. Some extreme LCS and LCF examples for a binary alphabet.

3 Lower Bounds: Markov Chains

The lower bounds are based on the work by Deken [Dek79] and Dančík & Paterson [PD94, Dan94]. They present a finite automaton that models an algorithm which finds a common subsequence (CS) on two infinite strings (tapes). By analyzing the associated Markov chain, a bound on the expected length of the LCS is found. The same idea can be applied to the LCF problem. A complete exposition of this section appears in [BYS95].

Dančík and Paterson use an automaton that alternatively reads from each one of the two unbounded tapes. We read at the same time from both strings, allowing the possibility of applying some symmetry rules which reduce the number of states. Informally, when reading a new pair of symbols of an alphabet Σ of size k with symbols $\{0, 1, \cdots, k-1\}$, the automaton outputs some matches that increase the CS and computes a new state based on the symbols not yet used. Therefore, at this point, all information about the past has been lost. So, we obtain a lower bound, because potentially, a longer CS (the LCS) could have been obtained looking at the complete strings. Nevertheless, the fact that we only have to look at the current state and the future, simplifies the problem by applying the following rules. Consider that each state $s \in S$ is identified by two strings $[a, b]$ which are the symbols not yet used in each tape, then:

1. We force that $|a| \geq |b|$. If it is not true, we just switch the two tapes and the behavior of the automaton is the same. This is only true because the contents of the tape are random and the symbols uniformly distributed.

2. We force that $a < b$ lexicographically on their first $\min(|a|, |b|)$ symbols. We do that by exchanging symbols. If $a[1]$ is not 0, we exchange in a and b all the occurrences of $a[1]$ with 0 and vice versa. The same thing can be done with b. If $b[1] > 1$ then we exchange in a and b all the occurrences of $b[1]$ with 1 and vice versa. This is valid because the symbols are indistinguishable and uniformly distributed.

These two rules diminish approximately by a factor of $2k^2$ the possible number of states that a machine like this can generate, by using classes of equivalence between states. Rule 2 can be extended recursively to $a[2]$, by permuting $a[2]$ with 2 if $a[2] > 2$, etc. We have done that for larger k, up to $k-1$ characters, reducing for every exchange the number of states by a factor of k. This symmetry is used in a similar way in [Dan94].

Formally, our CSS machine is a tuple (S, δ, O) where S is a set of states, δ is the transition function which given a state s and a pair of symbols gives the new state $(s' \leftarrow \delta(s, [x, y]))$, and O is the output function which given a state s and a pair of symbols $[x, y]$, returns the length of the chosen CS for that transition (this is explained later). The expected behavior of a CSS machine can be modeled by a strongly connected Markov chain (no absorbing states), where the probability of transition from one state to another state is the probability of the input symbol pair associated to that transition $(1/k^2)$. In the limit, the probability of being in a given state converges to the solution of

$$\mathbf{T}\vec{p} = \vec{p}\ , \quad \sum_i p_i = 1$$

where $\mathbf{T}$ is the probability transition matrix and $\vec{p}$ is the steady state probability vector [CM65]. After these probabilities are obtained, a lower bound on γ_k is given by

$$\gamma_k \ \geq \ \sum_{s \in S} p_s \sum_{[x,y] \in \Sigma \times \Sigma} \frac{O(s, [x, y])}{k^2}$$

CSS machines can be produced automatically as shown in [Dan94]. In our case we have a different production algorithm. The idea is that given a CSS machine $M(S, \delta, O)$, we select a subset of m states U_m from S and we expand those states. Expanding a state s means to concatenate all possible pairs of symbols to s, obtaining k^2 states. We normalize each of those states by applying rules 1 and 2 defined above. That is, all the transitions of s go to these states. Of those, some of them are new. Let S' be the set of new states. For each $s' \in S'$, we compute all the possible transitions as before, but we impose the condition that the states generated by s will have at most the same number of symbols of s'. If we have a larger number of symbols, we drop one or two symbols (we choose to delete the symbols with smaller frequency). If we produce new states, we add them to S' marking s' as expanded. The condition above implies that at some point all states in S' have been expanded, obtaining a new CSS machine M'. All states that have been expanded plus the states of M, form M'(see Figure 2).

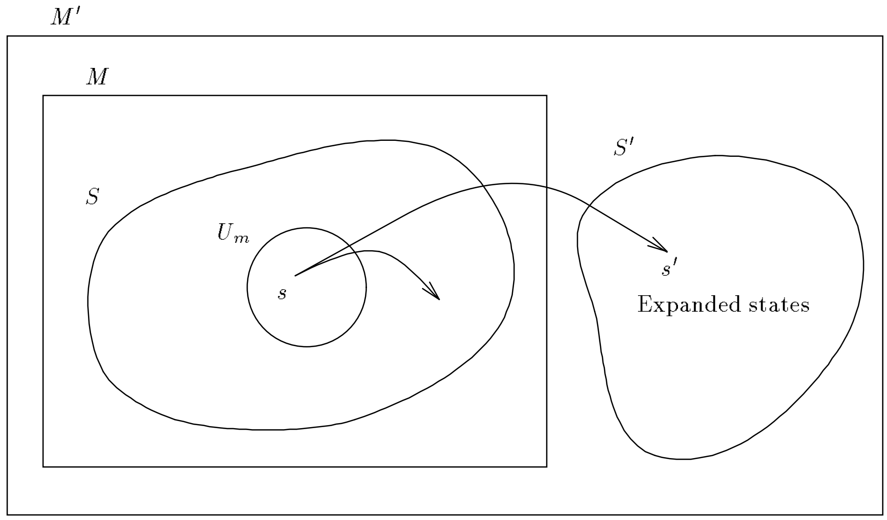

Fig. 2. Production process.

We can repeat this process several times to obtain larger and larger CSS machines, starting with the empty state $[\lambda, \lambda]$ where λ denotes the empty string.

There are several possibilities to generate the next state in a transition. We tried several ways to do it and the most successful one was the following. Given a state $s = [a, b]$, and a pair of symbols $[x, y]$, the next state is given by $s' = [a', b']$ such that $ax = ua'$ and $by = vb'$ where u and v are the strings that maximize

$$\frac{\ell cs(u, v)}{|u| + |v|}$$

if $\ell cs(a, b) > 0$. If there is more than one candidate we minimize over $|u| + |v|$. Otherwise, if $\ell cs(a, b) = 0$, we use $u = a[1]$. In this case, for v, we use $v = \lambda$ if $|a| > |b|$ or $|b| = 0$; else $v = b[1]$. This can be seen as a heuristic that locally maximizes γ_k by using the fewest possible number of characters. In practice, most of the time the cut u, v will happen on the "best" first match from left to right. Note that it may happen that $a[1] = b[1]$ in opposition to [Dan94] where they force the starting symbols to be different.

Figure 3 shows the basic CSS machine for general k for the LCS case when applying the production algorithm once starting from the empty state and using $m = 1$. The output function is shown between parenthesis.

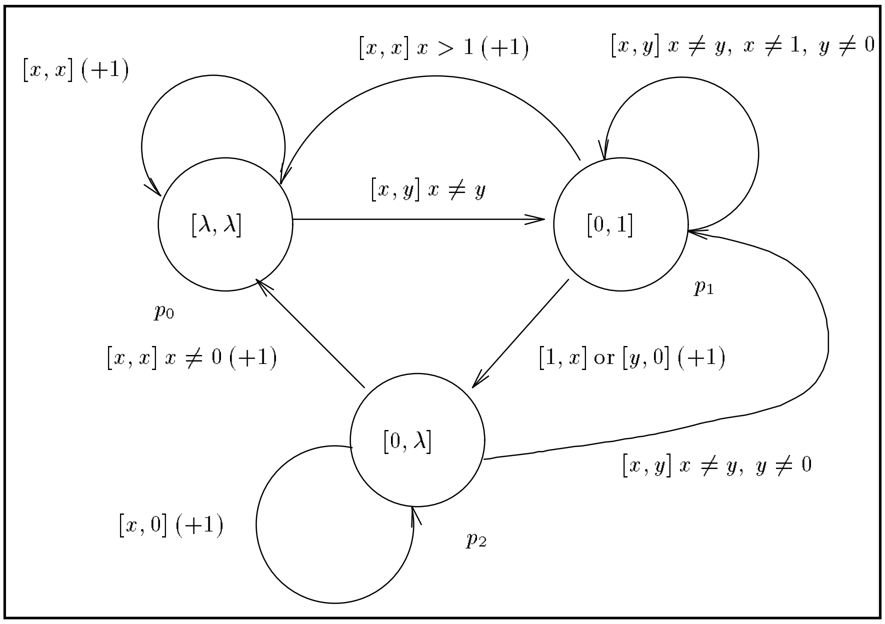

Fig. 3. CSS example for LCS.

The transition probability matrix of this example is

$$\mathbf{T} = \begin{bmatrix} 1/k & 1-1/k & 0 \\ (k-2)/k^2 & ((k-1)^2-(k-2))/k^2 & (2k-1)/k^2 \\ (k-1)/k^2 & (k-1)^2/k^2 & 1/k \end{bmatrix}$$

and the steady state probabilities are

$$p_0 = \frac{k^2-1}{D}\ p_1 = \frac{k^2(k-1)}{D},\ p_2 = \frac{k(2k-1)}{D},$$

where $D = k^3 + 2k^2 - k - 1$. For this automaton we have

$$\gamma_k \geq \frac{p_0}{k} + \frac{3(k-1)p_1}{k^2} + \frac{(2k-1)p_2}{k^2} = \frac{3k^2-k-1}{k^3+2k^2-k-1} = \frac{3}{k} + O(k^{-2})$$

For $k=2$ we obtain $\gamma_2 \geq 9/13 \approx 0.6923$.

In the production algorithm we have left open the question as to how to select U_m. Here, the number of states m to be expanded and the selection procedure is not fixed. In [Dan94] a next state is selected by "looking ahead" on the random input and chosing the transition where on average a longer CS is lost. Although this might be the best selection procedure, looking ahead can be computationally very expensive. They do it only for $k=2$ using the average of all possible strings of length 6. This is not practical for $k>2$ as the number of look ahead strings grows very fast. For that reason, we tried different heuristic cost functions associated with a state s. The one that gave the best results was to expand the states with largest expected output, that is:

$$Cost(s) = p_s \sum_{[x,y]\in\Sigma\times\Sigma} O(s,[x,y])$$

So, the selection procedure chooses the m states with largest $Cost$ to obtain U_m. For small k we used m between 2 and 10 to speed up the growing rate of the CSS machine. For larger k, $m = 1$ was enough, as the number of states grows exponentially.

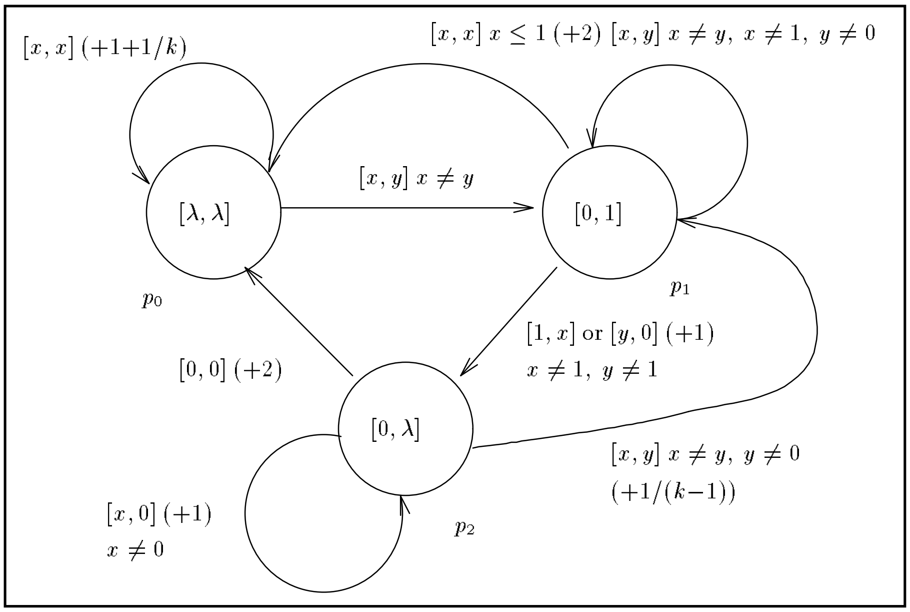

Fig. 4. CSS example for LCF.

The CSS machine for the LCF problem is given in Figure 4 for the case $m = 1$. We can further improve this automaton by noticing that in states 0 and 2, the previous event is always a match. So, if one of the new symbols is equal to the previous match, we can increase the LCF by 1. This has been considered in the output by adding the adequate terms which are a function of k. So, we have the following transition matrix

$$\mathbf{T} = \begin{bmatrix} 1/k & 1-1/k & 0 \\ 1/k & ((k-1)^2-(k-2))/k^2 & (2k-3)/k^2 \\ 1/k & (k-1)^2/k^2 & (k-1)/k^2 \end{bmatrix}$$

and we obtain

$$f_k \geq \frac{(k+1)p_0}{k^2} + \frac{(3k-1)}{k^2}(p_1+p_2) = \frac{3k^2-3k+2}{k^3}$$

which for $k = 2$ gives $f_2 \geq 1$.

The generation algorithm described has been implemented using the Maple symbolic algebra system [CGG+91]. Table 1 shows the lower bounds obtained so far by using our CSS machines up to 2000 states for the LCS and LCF problem.

k	Our γ_k	Previous γ_k	f_k (New)
	Lower bound	[PD94, Dan94]	Lower bound
2	0.75796	0.77391	1.41031
3	0.63376	0.61538	1.03554
4	0.55282	0.54545	0.83356
5	0.50952	0.50615	0.67948
6	0.46695	0.47169	0.56400

Table 1. New lower bounds for LCS and LCF.

4 Upper Bounds: Kolmogorov Complexity

The original goal of Kolmogorov complexity was to have a quantitative measure of the complexity of a finite object. Kolmogorov and others had the following idea: the regularities of an object can be used to give short descriptions of it; on the other hand, if an object is highly non-regular, or random, there should be no way of describing it that is much shorter than giving the full object itself. To formalize this notion, we first encode discrete objects as strings, as is customary in the theory of computation. Second, we want to have descriptions that can be handled algorithmically, so we identify descriptions with "programs for a sufficiently powerful model of computation".

Fix a Universal Turing Machine U whose input alphabet is $\{0, 1\}$ and output alphabet is Σ. The Kolmogorov complexity of a string $x \in \Sigma^\star$ is the minimum length of a program that makes U generate x and stops.

Observe that this definition seems to depend on the choice of the Universal Turing Machine. However, it can be shown that changing the machine only affects this measure of complexity by an additive constant.

Strings whose Kolmogorov complexity is equal, or close to, their length are called Kolmogorov-random. These are strings that cannot be compressed algorithmically.

As there are at most $2^n - 1$ binary "programs" of length $n-1$ or less, clearly there is some string of length n whose Kolmogorov complexity is at least n. A slight generalization of this counting argument gives that for every c and n, there are at most 2^{n-c} strings in Σ^n having Kolmogorov complexity $\leq n - c$.

For c even a small constant, this amounts to say that most strings, all but a fraction of 2^{-c}, are almost random: they cannot be compressed by more than c bits.

Many combinatorial properties have simple proofs via this prepackaged counting argument. Suppose that we want to show that property $P(x)$ holds for some string x. Take a Kolmogorov-random string x. Assume that $P(x)$ is false; show that this gives a way to describe x concisely. This is a contradiction. In fact, this argument usually gives proof that $P(x)$ holds with high probability, as the majority of strings are Kolmogorov random up to small constants.

For example, $P(x)$ could be some static property of x, such as "the difference between zeros and ones in x is at most $2\log|x|$"[4]; or a dynamic property such as "algorithm A takes time at most $5|x|$ on input x". In fact, several lower bounds on the (worst-case and expected) running time of algorithms have been proved using Kolmogorov complexity [LV93].

To apply this kind of argument to the case of LCS, observe that if two n-bit strings have a very long LCS (i.e., close to n bits), these two strings are in some sense very similar: knowing one of them gives away a lot of information about the other. Intuitively, if two strings are mutually random, knowing one of them should give essentially zero information to build the other. This must be true, in particular, if the two strings are obtained by chopping a Kolmogorov random string of $2n$ bits into two n-bit pieces. This argument is given in [LV93, Exercise 6.12, p.343], though in fact they only do it for $k = 2$.

We formalize this argument for general alphabets Σ: just bear in mind that we can identify strings of length n over k letters with binary strings of length $n\log k$.

We will determine γ such that $\ell cs(x,y) \leq \gamma n$ for Kolmogorov random strings x and y. Then averaging over all strings we obtain $EL_n^{(k)} \leq \gamma_k n + O(1/n)$. Indeed, let A be the set of words xy ($x, y \in \Sigma^n$) that have Kolmogorov complexity at least $(2n - 3\log n)\cdot\log k$. See that all but a fraction $O(1/n^3)$ of strings have this property. Then

$$\begin{aligned}
EL_n^{(k)} &= 1/k^{2n}\left[\sum_{xy\in A}\ell cs(xy) + \sum_{xy\notin A}\ell cs(xy)\right] \\
&\leq 1/k^{2n}\left[\sum_{xy\in A}\gamma n + \sum_{xy\notin A} n\right] \\
&\leq 1/k^{2n}\left[k^{2n}(1-O(1/n^3))\gamma n + k^{2n}O(1/n^3)n\right] \\
&= (1+O(1/n^2))\gamma n.
\end{aligned}$$

Assume $\ell cs(x,y) = \gamma n$. Clearly we can obtain xy if we have the following information:

- The values of n and γn.
- The LCS of x and y: $LCS(x,y)$.
- A description of the letter positions of x and y that give $LCS(x,y)$.
- The sequence of letters of x that do not belong to $LCS(x,y)$.

[4] All logarithms in this paper are in base 2.

– The sequence of letters of y that do not belong to the $LCS(x, y)$.

Formally, there is a fixed program (independent of n, x, and y) that, given this information, makes the Universal Turing Machine produce xy. As xy is random, the length of writing down this information in bits, plus the size of this program, must be at least $(2n - 3\log n)\log k$. Let us estimate the bit-length of each part.

The values of n and γn can be given in $2\log n$ bits each. By assumption, $LCS(x, y)$ can be encoded in $(\gamma n)\log k$ bits. The bits necessary to specify the letter positions is the log of the number of position sets that correspond to LCS's of two strings. Call this number $I_{n,\gamma}$.

For the last item, we use the following. A pair of strings may have several LCS's. We take as a representative that one with a lexicographically smallest set of positions: that is, if there are two choices for matching a letter we match it with the lowest index. Then, for every letter not in the LCS we can discard one out of k possibilities: if adjacent letters from positions i to j of x are not in the LCS, but letter $j+1$ is, we know that $x[k] \neq x[j+1]$, for any $i \leq k \leq j$. Hence, the $(1-\gamma)n$ letters of x not in the LCS can be encoded given as a string of length $(1-\gamma)n$ over an alphabet with $k-1$ letters, and similarly for y. In particular, for $k = 2$, this information is empty.

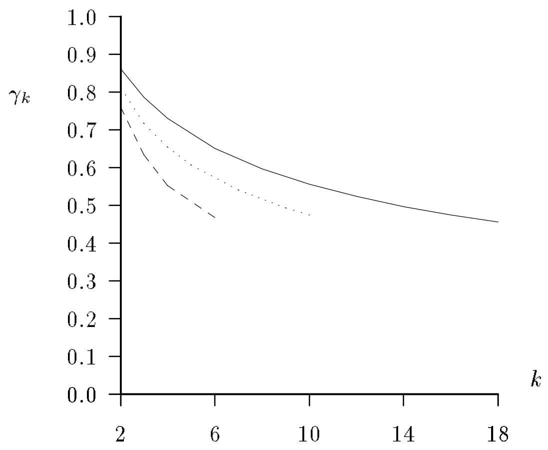

Fig. 5. Lower and upper bounds on γ_k for each alphabet size k. In between we show experimental results for $n = 100,000$.

Adding up, we obtain the equation

$$4\log n + \gamma_k n\log k + \log I_{n,\gamma_k} + 2(1-\gamma_k)n\log(k-1) \geq (2n - 3\log n)\log k$$

Dividing the equation by n, all sublinear terms vanish asymptotically, so we obtain:

$$\frac{\log I_{n,\gamma_k}}{n} + 2(1-\gamma_k)\log(k-1) \geq (2-\gamma_k)\log k. \quad (1)$$

A first upper bound on I_{n,γ_k} is the number of all subsets of $\{1 \ldots n\}$ with $\gamma_k n$ elements, squared (once for choosing in x, times the choice for y). By Stirling's approximation, $\log \binom{n}{\gamma_k n} = nH(\gamma_k)(1 + o(1))$, where $H(x) = -x\log(x) - (1 - x)\log(1 - x)$ is the binary entropy function. So we obtain the equation

$$2H(\gamma_k) + 2(1 - \gamma_k)\log(k - 1) \geq (2 - \gamma_k)\log k$$

For every k, solving this equation numerically gives a feasible range for γ_k. For example, for $k = 2$ it gives $0.282 \leq \gamma_2 \leq 0.867$. Figure 5 plots the values of γ_k up to $k = 18$, as well as experimental results for $n = 100,000$ (average taken over ten trials). Table 2 gives some exact values. By taking the limit on k, we obtain the already known result $\gamma_k \leq e/\sqrt{k}$.

For $k = 2$ this is the result obtained in [LV93]. We obtain a better bound for $k = 2$ by estimating more accurately the number of positions I_{n,γ_k}.

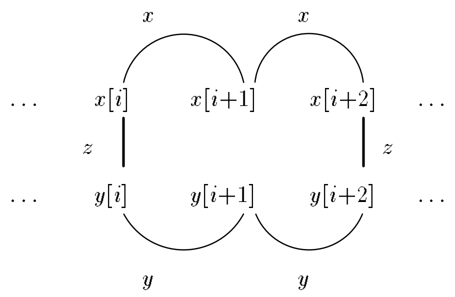

Fig. 6. Forbidden case for an LCS with $k = 2$, and counting variables used.

Consider the example given in Figure 6. If letters $x[i + 1]$ and $y[j + 1]$ are equal, we can match them and obtain a longer common sequence. If they are different, one of them equals $x[i + 2] = y[i + 2]$, so we can match it with either $x[i + 2]$ or $y[j + 2]$ and obtain a lexicographically smaller set of positions. So we have to count sets of positions that do not leave gaps simultaneously on the upper and lower strings.

As we will take the log of the number of strings divided by n for large n, we disregard smaller terms such as leading polynomials, etc., without further notice. In particular, we count only those strings that end with a match; it is not hard to see that this does not affect the base of the exponential.

To count the number of strings in the language defined, we use generating functions. Let $G(x, y, z)$ be

$$G(x, y, z) = \sum_{i,j,\ell} G_{i,j,\ell} x^i y^j z^\ell$$

where $G_{i,j,\ell}$ is the number of LCSs which have ℓcs of length ℓ with $i{+}1$ symbols in the upper string and $j + 1$ symbols in the lower string. That is, x is a symbolic variable associated to movements in the upper string, y to movements in the lower string, and z counts the edges between both strings (it may seem awkward to count movements and edges separately, but this makes possible to use the same approach for the LCF). The counting model is depicted in Figure 6. So, we are interested in $G_{n-1,n-1,n\gamma}$.

In our case we have,

$$G(x,y,z) = \left(\frac{1}{1-y} + \frac{x}{1-x}\right) yxzG(x,y,z) + 1 = \frac{1}{1 - \left(\frac{1}{1-y} + \frac{x}{1-x}\right) xyz}$$

That is, all strings are obtained by all possible ways to have zero or more y's ($1/(1-y)$) or zero or more x's, not counting twice the case of no letters in both strings ($1/(1-x) - 1$) and then a match xyz; concatenated with a string of the same form, that is $G(x,y,z)$. Then

$$G_\ell(x,y) = (xy)^\ell \left(\frac{1}{1-y} + \frac{x}{1-x}\right)^\ell = \sum_i \binom{\ell}{i} \frac{x^{i+\ell} y^\ell}{(1-x)^i (1-y)^{\ell-i}}$$

and

$$G_{m_1,m_2,\ell} = \sum_i \binom{\ell}{i} \binom{m_1 - \ell - 1}{i-1} \binom{m_2 - i - 1}{\ell - i - 1}$$

which when expressed in terms of the original n becomes

$$G_{n-1,n-1,\ell} = \sum_i \binom{\ell}{i} \binom{n-\ell}{i-1} \binom{n-i}{\ell - i - 1}$$

We do not need the exact solution to the above sum, just its logarithm divided by n, for large n. Call $M_{m,\ell}$ the maximum term of the summation. Then we have

$$\begin{array}{rcl} M_{n,\ell} \leq & G_{n,\ell} & \leq \ell M_{n,\ell} \\ \log(M_{n,\ell})/n \leq & \log(G_{n,\ell})/n & \leq \log(M_{n,\ell})/n + O\left(\frac{\log n}{n}\right) \end{array}$$

which shows that the larger term dominates the result. Moreover, we can maximize the logarithm of the term and use Stirling as before. Let $i = wn$, take the logarithm of the term i of the sum, divide by n and maximize with respect to w. We obtain that the maximum is reached for

$$w(\gamma) = \frac{2 - \gamma - \sqrt{5\gamma^2 - 8\gamma + 4}}{2}$$

that satisfies the constraints of the sum, namely $0 \leq w(\gamma) \leq \min(\gamma, 1-\gamma)$. By using this maximum term instead of the whole sum, and using the asymptotic formula $\log \binom{\alpha n}{\beta n} = \alpha n H(\beta/\alpha) + O(\log n)$, we have

$$\gamma H(w(\gamma)/\gamma) + (1-\gamma) H(w(\gamma)/(1-\gamma)) + (1-w(\gamma)) H((\gamma - w(\gamma))/(1-w(\gamma))) \geq 2 - \gamma$$

whose numerical solution is

$$\gamma_2 \leq 0.86019$$

which is still larger than what other more complicated theoretical models provide [Dan94], although quite close. Also, with this technique it is possible to obtain asymptotic results on k, which are not possible with ad-hoc methods.

Let us now consider the longest common forest problem. The LCF allows a better letter representation, since in this case not only each not connected letter must be different than that of the next alignment. The letters corresponding to each tree of the forest must be different than that of the next tree (otherwise we could join both trees). Hence, we need $\log(k-1)$ bits for all letters (connected and not connected), except the first one. For example, we need only one bit for $k=2$. Therefore, our inequality is

$$\frac{\log I_{n,f_k}}{n} + (2-f_k)\log(k-1) \geq 2\log k \tag{2}$$

The next step is to obtain a bound for I_{n,f_k}, the number of configurations for the forest. In this case, a single letter can be matched to many, so we drop the requirement for at least one gap between two edges. However, not both gaps can be zero. Hence,

$$G(x,y,z) = \left(\frac{1}{(1-x)(1-y)} - 1\right) zG(x,y,z) + 1 = \frac{1}{1-\left(\frac{1}{(1-x)(1-y)}-1\right)z}$$

Computing the inverse in z we have

$$\begin{aligned} G_\ell(x,y) &= \sum_\ell \frac{(x+y-xy)^\ell}{((1-x)(1-y))^\ell} \\ &= \sum_{i,j} (-1)^{\ell-i-j} \binom{\ell}{i}\binom{\ell-i}{j} \frac{x^i y^j (xy)^{\ell-i-j}}{((1-x)(1-y))^\ell} \end{aligned}$$

Now we invert in x and y to get

$$\begin{aligned} G_{m_1,m_2,\ell} &= \sum_{i,j} (-1)^{\ell-i-j} \binom{\ell}{i}\binom{\ell-i}{j}\binom{m_1+j-1}{\ell-1}\binom{m_2+i-1}{\ell-1} \\ &= \sum_i (-1)^{\ell+i}\binom{\ell}{i}\binom{m_2+i-1}{\ell-1}\sum_j(-1)^j\binom{\ell-i}{j}\binom{m_1+j-1}{\ell-1} \\ &= \sum_i (-1)^{\ell+i}\binom{\ell}{i}\binom{m_2+i-1}{\ell-1}(-1)^{\ell-i}\binom{m_1-1}{i-1} \end{aligned}$$

and by expressing it in terms of the original n we have

$$G_{n-1,n-1,\ell} = \sum_i \binom{\ell}{i}\binom{n+i}{\ell-1}\binom{n}{i-1}$$

Using the same maximizing technique as before ($i = wn$), we have

$$w(f) = \frac{-1+\sqrt{1+4f}}{2}$$

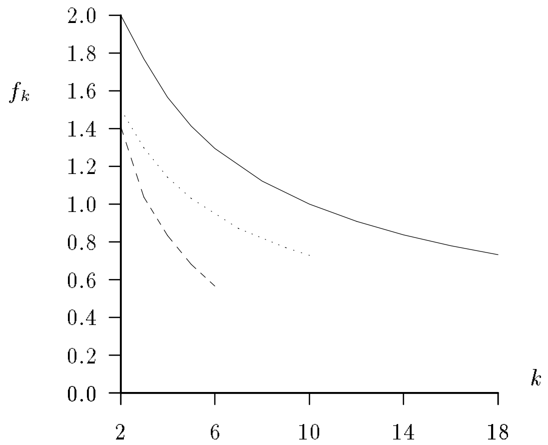

Fig. 7. Lower and upper bounds for f_k, for each alphabet size k. In between we show experimental results for $n = 100,000$.

This maximum value for $i = w(f)n$ is always in the bounds of the summation (i.e. $\max(f-1,0) \leq w(f) \leq \min(f,1)$). Then, we have

$$fH(w(f)/f)+(1+w(f))H(f/(1+w(f))+H(w(f)) \geq 2\log k-(2-f)\log(k-1) .$$

We can now numerically solve this inequality for each alphabet size k. Figure 7 plots the values of f_k up to $k = 18$ as well as experimental results for $n = 100,000$ (average taken over ten trials), and Table 2 shows some exact values. These are the first theoretical upper bounds for the LCF problem. Taking the limit on k, we obtain

$$f_k \leq \frac{e}{\sqrt{k}} \; + \; O(1/k)$$

k	Our γ_k Upper bound	Previous γ_k [PD94, Dan94]	f_k (New) Upper bound
2	0.86019	0.83763	2.00000
3	0.78647	0.76581	1.76704
4	0.72971	0.70824	1.56594
5	0.68612	0.66443	1.41289
6	0.65098	0.62932	1.29384
7	0.62172	0.60019	1.19855
8	0.59676	0.57541	1.12033
9	0.57507	0.55394	1.05478
10	0.55597	0.53486	0.99890
15	0.48538	0.46462	0.80753

Table 2. Upper bounds for LCS and LCF.

Acknowledgements

Some ideas for this work originated while the second author was visiting the University of Chile in Santiago and attending the XV Conference of the Chilean CS Society (SCCC) in Arica. He is grateful to Eric Goles and Martín Matamala for inviting him to the first, and to the SCCC and particularly Ricardo Baeza-Yates for inviting him to the second.

References

[BYS95] R. Baeza-Yates and R. Scheihing. New lower bounds for the expected length of longest common subsequences and forests. In *XV International Conference of the Chilean Computer Science Society*, pages 48–58, Arica, Chile, November 1995.

[CGG+91] B. Char, G. Geddes, G. Gonnet, B. Leong, M. Monagan, and S. Watt. *MAPLE V Language and Library Reference Manual.* Springer-Verlag, 1991.

[CM65] D. Cox and H. Miller. *The Theory of Stochastic Processes.* Chapman and Hall, London, 1965.

[CS75] V. Chvatal and D. Sankoff. Longest common subsequences of two random sequences. *Journal of Applied Probability*, 12:306–315, 1975.

[Dan94] V. Dančík. *Expected Length of Longest Common Subsequences.* PhD thesis, CS Dept, Univ. of Warwick, Warwick, UK, 1994.

[Dek79] J. Deken. Some limit results for longest common subsequences. *Discrete Mathematics*, 26:17–31, 1979.

[GBY91] G.H. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures - In Pascal and C.* Addison-Wesley, Wokingham, UK, 1991. (second edition).

[LV93] Ming Li and Paul Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications.* Springer-Verlag, New York, 1993.

[PD94] M. Paterson and V. Dančík. Longest common subsequences. In B. Rovan I. Privara and P. Ruzicka, editors, *19th MFCS'94*, LNCS 841, pages 127–142, Kosice, Slovakia, August 1994. Springer Verlag.

[PW93] P. Pevzner and M. Waterman. Generalized sequence alignment and duality. *Advances in Applied Mathematics*, 14:139–171, 1993.

[Ric95] Claus Rick. A new flexible algorithm for the longest common subsequence problem. In *CPM'95, 6th Annual Symposium on Combinatorial Pattern Matching*, Lecture Notes in Computer Science 937, pages 340–351, Espoo, Finland, 1995. Springer-Verlag.