10
10
Φ8cmcsc8

ON THE APPROXIMATION OF SHORTEST COMMON SUPERSEQUENCES AND LONGEST COMMON SUBSEQUENCES

TAO JIANG* AND MING LI†

Abstract. The problems of finding shortest common supersequences (SCS) and longest common subsequences (LCS) are two well-known **NP**-hard problems that have applications in many areas including computational molecular biology, data compression, robot motion planning and scheduling, text editing, etc. A lot of fruitless effort has been spent in searching for good approximation algorithms for these problems. In this paper, we show that these problems are inherently hard to approximate in the worst case. In particular, we prove that (i) SCS does not have a polynomial time linear approximation algorithm, unless **P** = **NP**; (ii) There exists a constant $\delta > 0$ such that, if SCS has a polynomial time approximation algorithm with ratio $\log^\delta n$, where n is the number of input sequences, then **NP** is contained in **DTIME**($2^{\text{polylog } n}$); (iii) There exists a constant $\delta > 0$ such that, if LCS has a polynomial time approximation algorithm with performance ratio n^δ, then **P** = **NP**. The proofs utilize the recent results of Arora *et al.* on the complexity of approximation problems.

In the second part of the paper, we introduce a new method for analyzing the average-case performance of algorithms for sequences, based on Kolmogorov complexity. Despite of the above non-approximability results, we show that near optimal solutions for both SCS and LCS can be found on the average. More precisely, consider a fixed alphabet Σ and suppose that the input sequences are generated randomly according to the uniform probability distribution and are of the same length n. Moreover, assume that the number of input sequences is polynomial in n. Then, there are simple greedy algorithms which approximate SCS and LCS with expected additive errors $O(n^{0.707})$ and $O(n^{\frac{1}{2}+\epsilon})$ for any $\epsilon > 0$, respectively.

Incidentally, our analyses also provide tight upper and lower bounds on the expected LCS and SCS lengths for a set of random sequences, solving a generalization of another well-known open question on the expected LCS length for two random sequences [2, 5, 24].

Key words. shortest common supersequence, longest common subsequence, approximation algorithm, **NP**-hardness, average-case analysis, random sequence.

AMS subject classifications. 68Q20, 68Q25

1. Introduction.

For two sequences $s = s_1 \ldots s_m$ and $t = t_1 \ldots t_n$, we say that s is a *subsequence* of t and equivalently, t is a *supersequence* of s, if for some $i_1 < \ldots < i_m$, $s_j = t_{i_j}$. Given a finite set of sequences, S, a *shortest common supersequence* (SCS) of S is a shortest possible sequence s such that each sequence in S is a subsequence of s. A *longest common subsequence* (LCS) of S is a longest possible sequence s such that each sequence in S is a supersequence of s.

These problems arise naturally in many practical situations. Researchers in many different areas have been trying for years to obtain partial solutions: dynamic programming when the number of sequences is constant, or, *ad hoc* algorithms when we do not care about absolute optimal solutions.

In artificial intelligence (specifically, planning), the actions of a robot (and human for that matter) need to be planned. A robot usually has many goals to be achieved. To achieve each goal the robot sometimes needs to perform a (linearly ordered) sequence of operations where identical operations in different sequences may be factored out and only performed once. Optimally merging these sequences of actions of the robot implies efficiency. Practical examples include robotic assembly lines [8] and metal-cutting in the domain of automated manufacturing [10, 15]. The essence of solutions to such problems are good approximation algorithms for the SCS problem. The SCS problem also has applications in text editing [24], data compression [27], and query optimization in database systems [25].

In molecular biology, a longest common subsequence (of some DNA sequences) is commonly used as a measure of similarity in the comparison of biological sequences [6]. Efficient algorithms for finding LCS could be very useful. Many papers in molecular biology have been written on this issue. We refer the readers to [6, 7, 26]. Other applications of the LCS problem include the widely used *diff* file comparison program [1], data compression and, syntactic pattern recognition [19].

For a long time, it has been well-known that the SCS and LCS problems (even on a binary alphabet) are all **NP**-hard [9, 21, 23]. For n sequences of length m, it is known that, by dynamic programming techniques, both SCS and LCS problems can be solved in $O(m^n)$ time; this result is independently due to many authors in computer science (e.g., [11, 29]) and biology. However, since the parameter m is usually extremely large in practice (*e.g.*, in computer text editing and DNA/RNA sequence comparison), the time requirement $O(m^n)$ is intolerable even for small to moderate n. There have been attempts to speed up the dynamic programming solution for LCS [12, 13]. The improved algorithms still run in $O(m^n)$ time in the worst case.

In the very first paper [21] that proves the **NP**-hardness of the SCS and LCS problems, Maier already asks for approximation algorithms. For the past many years, various groups of people, in the diverse fields of artificial intelligence, theoretical computer science, and biology have looked for heuristic algorithms to approximate the SCS and LCS problems, but so far, no polynomial time algorithms with guaranteed approximation bounds have been found.

In this paper, we show that it is indeed not surprising that the search for good approximation algorithms has been fruitless, because good approximations of these problems would imply that **P** = **NP**. Specifically,

* Supported in part by NSERC Operating Grant OGP0046613 and CGAT. Current address: Department of Computer Science and Systems, McMaster University, Hamilton, Ont. L8S 4K1, Canada. E-mail: jiang@maccs.mcmaster.ca

† Supported in part by the NSERC Operating Grant OGP0046506, ITRC, and CGAT. Current address: Department of Computer Science, University of Waterloo, Waterloo, Ont. N3L 3G1, Canada. E-mail: mli@math.waterloo.edu

1. No polynomial time algorithm can achieve constant approximation ratio for any constant for the SCS problem, unless $\mathbf{P} = \mathbf{NP}$. (The approximation ratio is the worst-case ratio between the approximate solution and the optimal solution.)
2. There exists a constant $\delta > 0$ such that, if SCS has a polynomial time approximation algorithm with ratio $\log^{\delta} n$, where n is the number of input sequences, then $\mathbf{NP}$ is contained in $\mathbf{DTIME}(2^{\mathrm{polylog}\ n})$.
3. There exists a constant $\delta > 0$ such that, if the LCS problem has a polynomial time approximation algorithm with performance ratio n^{δ}, then $\mathbf{P} = \mathbf{NP}$.

The above results assume an unbounded alphabet. Our proofs utilize the recent results of Arora *et al.* on the complexity of approximation problems [3]. An overview of the results in [3] that are of special interest to us will be given in the next section.

On the other hand, one should not be too discouraged by the non-approximability results above. Many heuristic algorithms for SCS and LCS seem to work well in practice. They are usually greedy-style algorithms and run very fast. These algorithms cannot always guarantee an optimal solution or even an approximately good solution, but they produce a satisfactory solution in most cases. In this paper, we try to provide a (partial) theoretical explanation by proving that both SCS and LCS can be indeed very well approximated *on the average*. Consider the SCS and LCS problems on a fixed alphabet Σ. Suppose that the input sequences are of length n, the number of input sequences is polynomial in n, and all sequences are *equally likely* and mutually independent. We show that some simple greedy algorithms approximate SCS and LCS with *expected additive errors* $O(n^{0.707})$ and $O(n^{\frac{1}{2}+\epsilon})$ for any $\epsilon > 0$, respectively. (The additive error is the difference between the approximate solution and optimal solution.) The algorithm for SCS is actually interesting and perhaps useful in practice, although the algorithm for LCS is somewhat trivial and impractical. Our analyses are based on a new average-case analysis method using Kolmogorov complexity.

It also turns out that such analyses enable us to also obtain a tight bound on the expected length of an LCS or an SCS of random sequences. Our results show that, over a fixed alphabet of size k, the expected length of an LCS (or an SCS) for n random sequences of length n is $\frac{n}{k} \pm n^{\frac{1}{2}+\epsilon}$ for any $\epsilon > 0$ (or $\frac{(k+1)n}{2} \pm O(n^{0.707})$, resp.). In contrast, the tight bound on the expected LCS length for two random sequences is a well-known open question in statistics [2, 5, 24].

In Section 2, we review the recent surprisingly fast development in the complexity theory of approximation. The hardness of approximating an SCS or an LCS is shown in Section 3. We analyze the average-case performance of some greedy algorithms in Section 4. Some concluding remarks are given in Section 5.

2. Recent Works on the Complexity of Approximation. Designing efficient approximation algorithms with good performance guarantees is not an easy task. This is due to the fact that the approximation of a large number of optimization problems is intractable. On the other hand, proving the intractability of an approximation problem can also be hard, essentially because the approximability properties are generally not preserved in a conventional polynomial time reduction [9]. Nevertheless, there have been some very significant developments in the last five years. Below we only discuss the work that will be needed for our results.

In 1988, Papadimitriou and Yannakakis defined a special reduction that preserves certain approximability properties [22]. Using this reduction and based on Fagin's syntactic definition of the class $\mathbf{NP}$, they introduced a class of natural optimization problems, $\mathbf{MAX\ SNP}$, which includes the Vertex Cover and Independent Set problems on bounded-degree graphs, Max Cut, various versions of Maximum Satisfiability, etc. It is known that every problem in this class can be approximated within *some* constant factor, and a polynomial time approximation scheme (PTAS) for any **MAX SNP**-complete problem would imply one for every other problem in the class. (A problem has a PTAS if for every fixed $\epsilon > 0$, the problem can be approximated within factor $1 + \epsilon$ in polynomial time.)

Recently, Arora *et al.* made some significant progress in the theory of interactive proofs [3]. As an application of their results, they showed that if any **MAX SNP**-hard problem has a PTAS, then **P** = **NP**, thus confirming the common belief that no **MAX SNP**-hard problem has a PTAS. Their results also show that, unless **P** = **NP**, the Largest Clique problem does not have a polynomial time approximation algorithm with performance ratio n^δ for some constant δ. Using these results and the graph product technique, Karger et al. are able to prove that longest paths cannot be approximated within any constant factor [14]. Very recently, Lund and Yannakakis showed that Graph Coloring cannot be approximated with ratio n^ϵ for some $\epsilon > 0$ and Set Covering cannot be approximated with ratio $c \log n$ for any $c < 1/4$ [20].

Before we leave this section, we recall the definition of the special reduction introduced by Papadimitriou and Yannakakis [22], used to show the **MAX SNP**-hardness of a problem. Suppose that Π and Π' are two optimization (*i.e.*, maximization or minimization) problems. We say that Π *L-reduces* (for *linearly reduces*) to Π' if there are two polynomial time algorithms f and g and constants $\alpha, \beta > 0$ such that, for any instance I of Π,

1. $OPT(f(I)) \leq \alpha \cdot OPT(I)$
2. Given any solution of $f(I)$ with weight w', algorithm g produces in polynomial time a solution of I with weight w satisfying $|w - OPT(I)| \leq \beta |w' - OPT(f(I))|$.

The following are two simple facts concerning L-reductions. First, the composition of two L-reductions is also an L-reduction. Second, if problem Π L-reduces to problem Π' and Π' can be approximated in polynomial time within a factor of $1 + \epsilon$, then Π can be approximated within factor $1 + \alpha\beta\epsilon$. In particular, if Π' has a PTAS, then so does Π.

A problem is **MAX SNP**-hard if every problem in **MAX SNP** can be L-reduced to it. Thus, by the result of Arora *et al.*, a **MAX SNP**-hard problem does not have a PTAS, unless **P** = **NP**.

3. Non-approximability of SCS and LCS. In this section, we show that there do not exist polynomial time approximation algorithms for SCS and LCS with *good* performance ratios. The proof for LCS is a direct reduction from the Largest Clique problem. The proof for SCS is more involved. We first show that a restricted version of SCS in which every input sequence is of length 2 and every letter of the alphabet appears at most 3 times in the input sequences is **MAX SNP**-hard. Thus this restricted version of SCS does not have a PTAS, assuming that **P** $\neq$ **NP**. Then we define the product of sets of sequences and relate the SCS of such a product to the SCS's of the components constituting the product. Finally, we demonstrate that a polynomial time constant ratio approximation algorithm for SCS would imply a PTAS for the restricted SCS, by blowing up instances using the product of sets of sequences.

3.1. Approximating LCS is Hard. THEOREM 3.1. *There exists a constant $\delta > 0$ such that, if the LCS problem has a polynomial time approximation algorithm with performance ratio n^δ, where n is the number of input sequences, then* **P** = **NP**.

Proof. We reduce the Largest Clique problem to LCS. Let $G = (V, E)$ be a graph and $V = \{v_1, \ldots, v_n\}$ the vertex set. Our alphabet Σ is chosen to be V.

Consider a vertex v_i and suppose that v_i is adjacent to vertices $v_{i_1}, \ldots, v_{i_q}$, where $i_1 < \ldots < i_q$. For convenience, let $i_0 = 0$ and $i_{q+1} = n+1$. Let p be the unique index such that $0 \leq p \leq q$ and $i_p < i < i_{p+1}$. We include two sequences:

$$x_i = v_{i_1} \cdots v_{i_p} v_i v_1 \cdots v_{i-1} v_{i+1} \cdots v_n$$

$$x'_i = v_1 \cdots v_{i-1} v_{i+1} \cdots v_n v_i v_{i_{p+1}} \cdots v_{i_q}$$

Let $S = \{x_i, x'_i | 1 \leq i \leq n\}$.

LEMMA 3.2. *The graph G has a clique of size k if and only if the set S has a common subsequence of length k, for any k.*

Proof. The "only if" part is clear. To prove the "if" part, let y be a common subsequence for S. If v_i appears in y, then the sequence x_i makes sure that all vertices on the left of v_i in y are adjacent to v_i in G and similarly the sequence x'_i ensures that that all vertices on the right of v_i in y are adjacent to v_i in G. Thus the vertices appearing in y actually form a clique of G. □

The proof is completed by recalling the result of Arora *et al.* that, unless **P** = **NP**, the Largest Clique problem does not have a polynomial time approximation algorithm with performance ratio n^δ on graphs with n vertices, for some constant $\delta > 0$ [3]. □

The above result shows that for some constant $\delta > 0$, there is no algorithm that, given a set S of n sequences, will find a common subsequence of length at least $OPT(S)/n^\delta$ in polynomial time. But it still remains open whether there might be some other $\delta < 1$ such that one can find a common subsequence of length at least $OPT(S)/n^\delta$ in polynomial time. We conjecture the answer to be in the negative.

It is also natural to measure the performance of an approximation algorithm for LCS in terms of the size of alphabet or the maximum length of its input sequences. Clearly the statement of Theorem 3.1 holds when n is replaced by these parameters.

We now consider the approximation of LCS on a fixed alphabet. Let the alphabet $\Sigma = \{a_1, ..., a_k\}$. It is trivial to show that LCS on Σ can be approximated with ratio k.

THEOREM 3.3. *For any set S of sequences from Σ, the following algorithm finds a common subsequence for S of length at least $OPT(S)/k$.*

Algorithm Long-Run.
Find maximum m such that a^m is a common subsequence of all input sequences, for some $a \in \Sigma$. Output a^m as the approximation of LCS.

It remains open whether LCS on a bounded alphabet is **MAX SNP**-hard. Although we believe that the answer is "yes", even when the alphabet is binary, we have not been able to establish an L-reduction from any known **MAX SNP**-hard problem. Observe that Maier's construction for the **NP**-hardness of LCS on a bounded alphabet [21] does not constitute an L-reduction. In his construction, an instance G of Vertex Cover is mapped to an instance S of LCS (or SCS) with the property that $OPT(S)$ is at least quadratic in $OPT(G)$.

Conjecture. LCS on a binary alphabet is **MAX SNP**-hard.

3.2. Restricted Versions of SCS and MAX-SNP-hardness. Maier proved the **NP**-hardness of SCS on an unbounded alphabet by reducing the Vertex Cover problem to SCS [21]. For any graph G of n vertices and m edges, the construction guarantees that G has a vertex cover of size t if and only the constructed instance of SCS has a common supersequence of length $2n + 6m + 8\max\{n, m\} + t$. It is easy to see that the reduction is actually linear if the graph G is of bounded degree. Since the Vertex Cover problem on bounded-degree graphs is **MAX SNP**-hard, so is SCS.

Let SCS(l, r) denote the restricted version of SCS in which each input sequence is of length l and each letter appears at most r times totally in all sequences. Such restricted problems have been recently studied by Timkovskii [28]. We will need the version SCS$(2, 3)$ later on to prove that SCS cannot be linearly approximated. It is known that SCS$(2, 2)$ can be solved in polynomial time and SCS$(2, 3)$ is **NP**-hard [28]. Obviously, SCS(l, r) can be approximated with ratio r. This is true since each letter appears only r times in total in an instance of SCS(l, r). Thus a plain concatenation already achieves an approximation ratio r.

THEOREM 3.4. *SCS$(2, 3)$ does not have a PTAS, unless* **P** = **NP**.

Proof. A polynomial time reduction from the Feedback Vertex Set problem on bounded-degree digraphs [9] to SCS$(2, 3)$ is given in [28]. Let $G = (V, E)$ be a digraph of degree 3. The reduction defines a set $S = \{uv | (u, v) \in E\}$. Clearly, S is an instance of SCS$(2, 3)$. It is shown that G has feedback vertex set of size t if and only if S has a common supersequence of length $|V| + t$.

It is easy to L-reduce Vertex Cover on bounded-degree graphs to Feedback Vertex Set on bounded-degree graphs, by replacing each edge in the instance of Vertex Cover with a directed cycle. The reduction from Feedback Vertex Set to SCS$(2, 3)$ is actually linear for the digraphs resulted from this construction, as the optimal feedback vertex set for these graphs is linear in $|V|$. Since the composition of two L-reductions is an L-reduction, we have an L-reduction from the Vertex Cover problem on bounded degree graphs to SCS$(2, 3)$. The theorem follows from the fact that Vertex Cover on bounded degree graphs is **MAX SNP**-hard. □

The status of the complexity of approximating SCS on a fixed alphabet is quite similar to that for LCS. We can show that SCS on a fixed alphabet has a trivial constant ratio approximation. But we do not know if the problem is **MAX SNP**-hard. Again, observe that the reduction in Maier's original proof of the **NP**-hardness for SCS on a bounded alphabet is not linear.

THEOREM 3.5. *Let Σ be an alphabet of k letters. For any set S of sequences from Σ, we can find a common supersequence for S of length at most $k \cdot OPT(S)$ in polynomial time.*

Proof. Let l_{max} be the maximum length of input sequences in S. Then $(a_1 \cdots a_k)^{l_{max}}$ is a common supersequence for S satisfying the length requirement. □

Conjecture. SCS on a binary alphabet is **MAX SNP**-hard.

3.3. The Product of Sets of Sequences. First, we extend the operation "concatenation" to sets of sequences. Let X and Y be two sets of sequences. Define the concatenation of X, Y, denoted $X \cdot Y$, as the set $\{x \cdot y | x \in X, y \in Y\}$. For example, if $X = \{abab, aabb\}$ and $Y = \{123, 231, 312\}$, then

$$X \cdot Y = \{abab123, aabb123, abab231, aabb231, abab312, aabb312\}.$$

The following lemma is quite useful in our construction.

LEMMA 3.6. *Let $X = X_1 \cdot X_2 \cdots X_n$. Suppose that y is a supersequence for X. Then there exist $y_1, y_2, \ldots, y_n$ such that $y = y_1 \cdot y_2 \cdots y_n$ and each y_i is a supersequence for X_i, $1 \le i \le n$.*

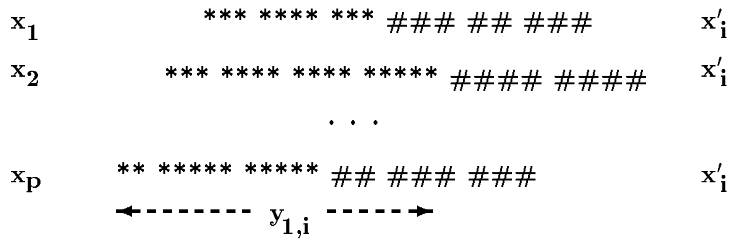

FIG. 1. *Finding $y_{1,i}$ in the supersequence y.*

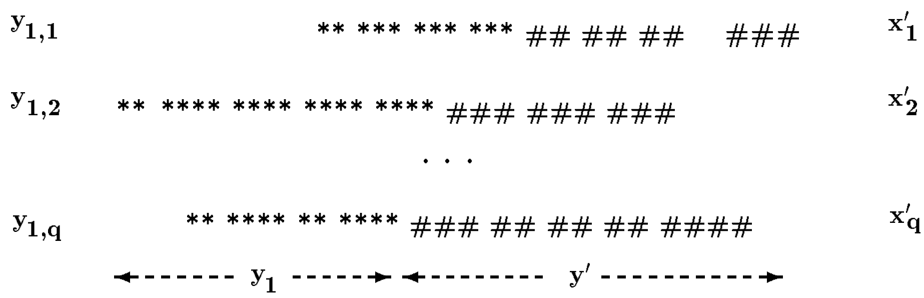

FIG. 2. *Finding y_1 in the supersequence y.*

Proof. Let $X_1 = \{x_1, \ldots, x_p\}$ and $X' = X_2 \cdots X_n = \{x'_1, \ldots, x'_q\}$. Fix a sequence x'_i in X'. Since the supersequence y contains $x_1 \cdot x'_i, \ \ldots, \ x_p \cdot x'_i$ as subsequences, it must contain a subsequence $y_{1,i} \cdot x'_i$, where $y_{1,i}$ is a supersequence for X_1. To see this, just consider the positions of the first letter of x'_i in the subsequences $x_1 \cdot x'_i, \ \ldots, \ x_p \cdot x'_i$ of y. The rightmost occurrence gives $y_{1,i}$. See Figure 1. In the figure, the $*$'s represent sequences $x_1, \ldots, x_p$ and the #'s represent the sequences $x'_1, \ldots, x'_q$. The sequences are aligned up according to their relative positions in y.

Thus, for each $1 \le i \le q$, we have a sequence $y_{1,i}$ such that $y_{1,i}$ is a supersequence for X_1 and $y_{1,i} \cdot x'_i$ is a subsequence of y. Now we consider the positions of the last letter of $y_{1,1}, \ldots, y_{1,q}$ in the subsequences $y_{1,1} \cdot x'_1, \ldots, y_{1,q} \cdot x'_q$ of y and partition y at the leftmost such position. Let the left part of y be y_1 and the right part be y'. Then clearly y_1 is a supersequence for X_1 and y' is a supersequence for X'. See Figure 2. In this figure, the $*$'s now represent sequences $y_{1,1}, \ldots, y_{1,q}$.

We can recurse on y' and Y' to obtain $y_2, \ldots, y_n$. $\square$

We are now ready to define the product of sets of sequences. The symbol $\times$ will be used to denote the product operation. We start by defining the product of single letters. Let Σ and Σ' be two alphabets, and $a \in \Sigma$, $b \in \Sigma'$ two letters. The product of a and b is just the composite letter $(a, b) \in \Sigma \times \Sigma'$. The product of a sequence $x = a_1 \cdot a_2 \cdots a_n$ and a letter b is $a_1 \times b \ \cdot \ a_2 \times b \ \cdots \ a_n \times b$. The product of a set $X = \{x_1, x_2, \ldots, x_n\}$ of sequences and a letter a is the set $\{x_1 \times a, x_2 \times a, \ldots, x_n \times a\}$. The product of a set X of sequences and a sequence $y = a_1 \cdot a_2 \cdots a_n$ is the set $X \times a_1 \ \cdot \ X \times a_2 \ \cdots \ X \times a_n$. Finally, let X and $Y = \{y_1, y_2, \ldots, y_n\}$ be two sets of sequences. The product of X and Y is the set $\bigcup_{i=1}^{n} X \times y_i$. For example,

$$(a,1)(a,1)(b,1)(b,1)(a,2)(a,2)(b,2)(b,2)(a,1)(a,1)(b,1)(b,1)$$
$$(a,1)(a,1)(b,1)(b,1)(a,2)(a,2)(b,2)(b,2)(a,1)(b,1)(a,1)(b,1)$$
$$(a,1)(a,1)(b,1)(b,1)(a,2)(b,2)(a,2)(b,2)(a,1)(a,1)(b,1)(b,1)$$
$$(a,1)(a,1)(b,1)(b,1)(a,2)(b,2)(a,2)(b,2)(a,1)(b,1)(a,1)(b,1)$$
$$(a,1)(b,1)(a,1)(b,1)(a,2)(a,2)(b,2)(b,2)(a,1)(a,1)(b,1)(b,1)$$
$$(a,1)(b,1)(a,1)(b,1)(a,2)(a,2)(b,2)(b,2)(a,1)(b,1)(a,1)(b,1)$$
$$(a,1)(b,1)(a,1)(b,1)(a,2)(b,2)(a,2)(b,2)(a,1)(a,1)(b,1)(b,1)$$
$$(a,1)(b,1)(a,1)(b,1)(a,2)(b,2)(a,2)(b,2)(a,1)(b,1)(a,1)(b,1)$$

$$(a,2)(a,2)(b,2)(b,2)(a,1)(a,1)(b,1)(b,1)(a,2)(a,2)(b,2)(b,2)$$
$$(a,2)(a,2)(b,2)(b,2)(a,1)(a,1)(b,1)(b,1)(a,2)(b,2)(a,2)(b,2)$$
$$(a,2)(a,2)(b,2)(b,2)(a,1)(b,1)(a,1)(b,1)(a,2)(a,2)(b,2)(b,2)$$
$$(a,2)(a,2)(b,2)(b,2)(a,1)(b,1)(a,1)(b,1)(a,2)(b,2)(a,2)(b,2)$$
$$(a,2)(b,2)(a,2)(b,2)(a,1)(a,1)(b,1)(b,1)(a,2)(a,2)(b,2)(b,2)$$
$$(a,2)(b,2)(a,2)(b,2)(a,1)(a,1)(b,1)(b,1)(a,2)(b,2)(a,2)(b,2)$$
$$(a,2)(b,2)(a,2)(b,2)(a,1)(b,1)(a,1)(b,1)(a,2)(a,2)(b,2)(b,2)$$
$$(a,2)(b,2)(a,2)(b,2)(a,1)(b,1)(a,1)(b,1)(a,2)(b,2)(a,2)(b,2)$$

FIG. 3. *The product of* $\{aabb, abab\}$ *and* $\{121, 212\}$.

if $X = \{aabb, abab\}$ and $Y = \{121, 212\}$, then $X \times Y$ contains the 16 sequences in Figure 3.

Note that if each sequence in X has length l_1 and each sequence in Y has length l_2, then $X \times Y$ contains $|Y| \cdot |X|^{l_2}$ sequences of length $l_1 \cdot l_2$. Thus, $X \times Y$ does not have polynomial size in general.

In this paper, we will only be interested in products in which the second operand has a special property. Let Y be a set of sequences with the following property: each sequence is of even length and every letter at an even position is *unique* in Y, *i.e.*, the letter appears only once (totally) in all sequences in Y. We will refer to such unique letters as *delimiters*. The following lemma relates the SCS of a product to the SCS's of its operands and is crucial to the next subsection.

LEMMA 3.7. *Let X and Y be sets of sequences. Suppose that Y has the above special property. Then $OPT(X \times Y) = OPT(X) \cdot OPT(Y)$. Moreover, given a supersequence for $X \times Y$ of length l, we can find in time polynomial in $|X \times Y|$ supersequences for X and Y of lengths l_1 and l_2, respectively, such that $l_1 \cdot l_2 \leq l$.*

Proof. Clearly $OPT(X \times Y) \leq OPT(X) \cdot OPT(Y)$. Suppose that z is a supersequence for $X \times Y$ of length l. We show how to find supersequences for X and Y of lengths l_1 and l_2, respectively, such that $l_1 \cdot l_2 \leq l$, in polynomial time. Let Σ be the alphabet corresponding to Y. For each letter $a \in \Sigma$, we call the product $X \times a$ an *a component*. The letters are divided into delimiters and non-delimiters. For convenience, call the non-delimiters *normal* letters. Our basic idea is to rearrange the supersequence z without increasing length such that each component appears in a consecutive region. Then we can "extract" the components and identify the desired supersequences for X and Y.

Using Lemma 3.6, we can extract from z a supersequence for each delimiter component as follows. Let $y = a_1 a_2 \cdots a_n$ be a sequence in Y. Consider the delimiter a_2. Since z is a supersequence for the product $X \times y = X \times a_1 \cdot X \times a_2 \cdots X \times a_n$, there must be $z_1, z_2, \ldots, z_n$ such that z_i is a supersequence for $X \times a_i$ and $z = z_1 \cdot z_2 \cdots z_n$. Now we look at z_2 and concentrate on the sequences in the a_2 component. We can rearrange z_2 such that the letters appearing in the a_2 component form a consecutive block, by shifting them to the right appropriately. Denote the new sequence z_2'. Since a_2 is unique in Y, $z_1 \cdot z_2' \cdots z_n$ is also a supersequence for $X \times Y$. This way we have extracted a supersequence for the a_2 component. Supersequences for other delimiter components can be extracted similarly. Note the above does not increase the length of the whole supersequence.

Call the final supersequence after the above process z'. So z' has the form $u_1 \cdot v_1 \cdot u_2 \cdot v_2 \cdots$, where each v_i is a supersequence for some delimiter component and u_i is a sequence of letters from some normal components. We can easily rearrange z' so each u_i actually becomes either nil or a supersequence for some normal component, by shifting the normal component letters to the rightmost possible position (stopped only by some relevant delimiter block). Thus we now have a supersequence of the form $u_1' \cdot v_1 \cdot u_2' \cdot v_2 \cdots$, where each u_i' is either nil or a supersequence for some normal component. Now the pattern $u_1' \cdot v_1 \cdot u_2' \cdot v_2 \cdots$, naturally defines a supersequence for Y. Let l_1 be the minimum length of the supersequences for the components (and thus X) and l_2 the length of the supersequence for Y. Then $l_1 \cdot l_2 \leq l$. □

3.4. SCS Has No Linear Approximation Algorithms. Our basic idea is to use the product operation to blow up a given instance of SCS. However, the product of sets of sequences cannot be performed in polynomial time, unless the the second operand contain sequences of bounded length. Thus we consider the restricted version SCS$(2,3)$. A nice thing about SCS$(2,3)$ is that for any instance S of SCS$(2,3)$, the total length of the sequences in S is at most $3 \cdot OPT(S)$. Thus, we can insert unique delimiters into the sequences as required in the previous subsection without affecting the **MAX SNP**-hardness. So, let SCS$(2,3)'$ denote the version whose instances are obtained from instances of SCS$(2,3)$ by inserting unique delimiters. Let S be an instance of SCS$(2,3)$ of total length n and S' be the corresponding instance of SCS$(2,3)'$. It is easy to see that S has a common supersequence of length t if and only if S' has a common supersequence of length $n+t$. Since $t = \theta(n)$, this forms an L-reduction from SCS$(2,3)$ to SCS$(2,3)'$, and hence SCS$(2,3)'$ is also **MAX SNP**-hard.

For any set S, S^k denotes the product of k S's. The next lemma follows from Lemma 3.7.

LEMMA 3.8. *Let $k \geq 1$ be a fixed integer. For any instance S of SCS$(2,3)'$, $OPT(S^k) = OPT(S)^k$. Moreover, given a supersequence for S^k of length l, we can find in polynomial time a supersequence for S of length $l^{1/k}$.*

Observe that, if $|S| = n$, then $|S^k| = n^{O(4^k)}$, since each sequence in S has length 4. Now we can prove the main result in this section.

THEOREM 3.9. *(i) There does not exist a polynomial time linear approximation algorithm for SCS, unless* $\mathbf{P} = \mathbf{NP}$. *(ii) There exists a constant $\delta > 0$ such that, if SCS has a polynomial time approximation algorithm with ratio $\log^\delta n$, where n is the number of input sequences, then* **NP** *is contained in* $\mathbf{DTIME}(2^{\text{polylog } n})$.

Proof. We only prove (i). The proof of (ii) is similar. The idea is to show that if SCS has a polynomial time linear approximation algorithm, then SCS$(2,3)'$ has a PTAS. Suppose that SCS has a polynomial time approximation algorithm with performance ratio α. For any given $\epsilon > 0$, let $k = \lceil \log_{1+\epsilon} \alpha \rceil$. Then by Lemma 3.8, we have an approximation algorithm for SCS$(2,3)'$ with ratio $\alpha^{1/k} \leq 1+\epsilon$. The algorithm runs in time $n^{O(4^k)}$, thus polynomial in n. This implies a PTAS for SCS$(2,3)'$. □

It is interesting to note that our non-approximability result for SCS is weaker than that of LCS (and the longest path problem in [14]). It seems to require new techniques to prove a stronger lower bound. The growth rate of $n^{O(4^k)}$ is too high. This is essentially due to the way we define the product of sets of sequences. If we can find a better way of taking products and lower the rate to something like $n^{O(k)}$, then the bound in (ii) can be strengthened to $2^{\log^\delta n}$, for any $\delta < 1$, as shown in [14] for the longest path problem.

4. Algorithms with Good Average-Case Performance. We have seen that the LCS and SCS problems are not only **NP**-hard to solve exactly, but also **NP**-hard to approximate. The approximation of these problems restricted to fixed alphabets also seems to be hard. In this section, we consider the average-case performance of some simple greedy algorithms for LCS and SCS and prove that these algorithms can find a *nearly optimal* solution in *almost all the cases*, assuming that all sequences are equally likely and the sequences are independent of each other. Note that, our probability model may not be realistic as in practice the sequences are usually *related* to each other and thus are not independent.

From now on, let $\Sigma = \{a_1, \ldots, a_k\}$ be a fixed alphabet of size k. For convenience, we will assume that the input is always n sequences over Σ, each of length n, although our results actually hold when the number of input sequences is polynomial in n. We prove that some remarkably simple greedy algorithms can approximate LCS and SCS with minor expected additive errors. Some of the technical results are actually quite interesting in their own rights. They give tight bounds on the expected length of an LCS or SCS of n random sequences of length n. It turns out that Kolmogorov complexity is a convenient and crucial tool for our analyses.

4.1. Kolmogorov Complexity. Kolmogorov complexity has been used in [8] as an effective method for analyzing the average-case performance of some algorithms for the SCS problem. It was also used recently by Ian Munro (see [17]) to obtain the average-case complexity of Heapsort, solving a long standing open question. We use Kolmogorov complexity as a tool here to analyze some combinatorial properties of random sequences which result in tight upper and lower bounds on the average-case performance of some algorithms for LCS and SCS problems. One can compare the usage of Kolmogorov complexity with the probabilistic method. In a sense, taking a Kolmogorov random string as the input is like taking an expectation as in the second moment method. A Kolmogorov random input has the random string properties and yet these properties hold with certainty rather than with some (high) probability. This fact greatly simplifies many proofs. To make the paper self-contained, we briefly review the definition and some properties of Kolmogorov complexity, tailored to our needs. For a complete treatment of this subject, see [17] or the survey [16].

Fix a universal Turing machine U with input alphabet Σ. The machine U takes two parameters p and y. U interprets p as a program and simulates p on input y. The Kolmogorov complexity of a string $x \in \Sigma^*$, given $y \in \Sigma^*$, is defined as

$$K_U(x|y) = \min\{|p| : U(p, y) = x\}.$$

Because one can prove an invariance theorem which claims that Kolmogorov complexity defined with respect to any two different universal machine differs by only an additive constant, we will drop the subscript U. In fact, for the purpose of our analysis, one fixed universal Turing machine is always assumed. Thus $K(x|y)$ is the *minimum* number of *digits* (*i.e.*, letters in Σ) in a description from which x can be effectively reconstructed, given y. Let $K(x) = K(x|\lambda)$, where λ denotes the null string.

By simple counting, for each $n, c < n$, and y, there are at least $|\Sigma|^n - |\Sigma|^{n-c} + 1$ distinct x's of length n with the property

$$K(x|y) \geq n - c. \tag{1}$$

We call a string x of length n *random* if

$$K(x) \geq n - \log n$$

where the logarithm is taken over base $|\Sigma|$. Sometimes, we need to encode x in a *self-delimiting* form $\bar{x}$, in order to be able to decompose $\bar{x}y$ into x and y. One possible coding for $\bar{x}$ may be $10L(x)10x$ where $L(x)$ is the binary representation of $|x|$ with each bit doubled. (Assume that $0, 1 \in \Sigma$.) For example, if $x = 0000000$, $|x| = 111$ in binary, and $L(x) = 111111$. This costs us about $2\log|x|$ extra bits. Thus, the self-delimiting representation $\bar{x}$ of x requires at most $|x| + 2\log|x| + 4$ digits [16]. Notice that our Kolmogorov complexity is a bit unconventional, since here we consider strings over the arbitrary fixed alphabet Σ instead of binary strings.

4.2. Longest Common Subsequences: the Average Case. We have shown that the LCS problem cannot be approximated with ratio n^δ for some $\delta > 0$ in polynomial time, unless $\mathbf{NP} = \mathbf{P}$. Thus no polynomial time algorithm can produce even approximately *long* common subsequences. However this claim only holds for the (probably extremely rare) worst cases. Here we would like to show that for random input sequences, the LCS problem can be approximated up to a small additive error.

THEOREM 4.1. *For an input set S containing n sequences of length n, the algorithm Long-Run approximates the LCS with an expected additive error $O(n^{\frac{1}{2}+\epsilon})$, for arbitrarily small $\epsilon > 0$.*

The proof is based on the following two lemmas which give a lower bound on performance of Long-Run and an upper bound on the length of an LCS, for a set of Kolmogorov random strings.

LEMMA 4.2. *Let $\epsilon > 0$ be any constant and x a string of length n. If some letter $a \in \Sigma$ appears in x less than $\frac{n}{k} - n^{\frac{1}{2}+\epsilon}$ times or more than $\frac{n}{k} + n^{\frac{1}{2}+\epsilon}$ times, then for some constant $\delta > 0$,*

$$K(x_i) \leq n - \delta n^{2\epsilon}.$$

Proof. In principle, this result can be proved using methods in [18]. By encoding each letter as a binary string of $\log_2 k$ bits, the results in [18] imply that that when $\log_2 k$ is an integer the lemma is true. To show the general case, we simply do a direct estimation as follows.

Suppose that for some letter $a \in \Sigma$, x contains only $d = \frac{n}{k} - n^{\frac{1}{2}+\epsilon}$ a's. There are only

$$\binom{n}{d}(k-1)^{n-d}$$

strings of length n with d occurrences of a. Taking a logarithm with base k would give us the number of digits to specify x_i. By an elementary estimation (using Stirling's formula), we can show that

$$\log_k \binom{n}{d} (k-1)^{n-d} \leq n - \delta n^{2\epsilon}, \tag{2}$$

for some $\delta > 0$. Thus $K(x_i) \leq n - \delta n^{2\epsilon}$. □

Now consider a fixed Kolmogorov random string x of length n^2. Cut x into n equal length pieces $x_1, \ldots, x_n$. It follows from Lemma 4.2 and the fact the randomness of x that, for each $a \in \Sigma$, a appears in each of $x_1, \ldots, x_n$ at least $\frac{n}{k} - O(n^{1/2+\epsilon})$ times, for any small $\epsilon > 0$. Thus

$$a^{n/k - O(n^{1/2+\epsilon})} \tag{3}$$

is a common subsequence of sequences $x_1, \ldots, x_n$. In the next lemma, we show that an LCS for $x_1, \ldots, x_n$ cannot really be much longer than the one in Formula (3).

LEMMA 4.3. *For any common subsequence s of $x_1, \ldots, x_n$,*

$$|s| < \frac{n}{k} + n^{\frac{1}{2}+\epsilon}. \tag{4}$$

Proof. For the purpose of a contradiction, suppose that

$$|s| \geq \frac{n}{k} + n^{\frac{1}{2}+\epsilon}.$$

We will try to compress x using s. The idea is to save n^δ digits for some $\delta > 0$ on each x_i.

Fix an x_i. We do another encoding of x_i. Let $s = s_1 s_2 \ldots s_p$, $p = |s|$. We align the letters in s with the corresponding letters in x_i greedily from left to right, and rewrite x_i as follows:

$$\alpha_1 s_1 \alpha_2 s_2 \ldots \alpha_p s_p x_i'. \tag{5}$$

Here α_1 is the longest prefix of x_i containing no s_1; α_2 is the longest substring of x_i starting from s_1 containing no s_2, and so on. x_i' is the remaining part of x_i after s_n. Thus α_j does not contain letter s_j, for $j = 1, \ldots, p$. That is, each α_j contains at most $k-1$ letters in Σ. In other words, $\alpha_j \in (\Sigma - \{s_j\})^*$.

We now show that x_i can be compressed, by at least n^δ digits for some $\delta > 0$, with the help of s. In fact, it is sufficient to prove that the prefix

$$y = \alpha_1 s_1 \alpha_2 s_2 \ldots \alpha_p s_p, \tag{6}$$

can be compressed by this amount.

Using s, we know which $k-1$ letters in Σ appear in α_i, for each i. Thus we can recode y as follows. For each i, if $s_i = a_k$, then do nothing. Otherwise, replace s_i by a_k, the last letter in Σ and in α_i, replace every a_k by s_i. We can convert this transformed string y' back to y, using s, by reversing above process.

Now this transformed string y' is also a string over Σ. But in y', letter a_k appears $\frac{n}{k} + n^{\frac{1}{2}+\epsilon}$ times, since $|s|$ is least this long. By Lemma 4.2, for some constant $\delta > 0$, we have

$$K(y') \leq |y'| - \delta n^{2\epsilon}/k.$$

But since from y' and s we can obtain y and hence, together with a self-delimiting form of x_i', obtain x_i. We conclude that

$$K(x_i|s) \leq n - \delta n^{2\epsilon}/k + O(\log n),$$

where the term $O(\log n)$ takes care of the extra digits required for the self-delimiting representation of x_i' and the description of above procedure.

We repeat this for every x_i. In total, we save $\Omega(n^{1+2\epsilon})$ digits to encode x. Thus,

$$K(x) \leq n^2 - \Omega(n^{1+2\epsilon}) + |s| + O(n \log n) < |x| - \log |x|.$$

Therefore, x is not random, and a contradiction! □

We are now ready to prove the theorem.

Proof of Theorem 4.1. Consider all possible inputs of n sequences of length n. For each such input, we concatenate the n sequences together to obtain one string. Only about $1/n^2$ of them are not random by Formula (1). That is, only $1/n^2$ of them do not satisfy $K(x) \geq |x| - \log |x|$. For all the others, The above lower and upper bounds apply, and the algorithm Long-Run produces a common subsequence that is at most $O(n^{\frac{1}{2}+\epsilon})$ shorter than the LCS, for any fixed $\epsilon > 0$. Observe that the worst-case error of Long-Run is $(k-1)n/k$. Thus, a simple averaging shows that the expected error of Long-Run is $O(n^{\frac{1}{2}+\epsilon})$, for any fixed $\epsilon > 0$.

Lemmas 4.2 and 4.3 actually give very tight upper and lower bounds on the expected LCS length of n random sequences of length n. We note in passing that the same problem for 2 sequences is still open and there is a large gap between the current best upper and lower bounds [2, 5, 24].

Corollary 4.4. *The expected length of an LCS for a set of n random sequences of length n is $\frac{n}{k} \pm n^{\frac{1}{2}+\epsilon}$ for any $\epsilon > 0$.*

4.3. Shortest Common Supersequences: the Average Case. In [8], the performance of the following algorithm for the SCS problem is analyzed.

Algorithm Majority-Merge

1. Input: n sequences, each of length n.
2. Set supersequence $s :=$ *null string*;
3. Let a be the majority among the leftmost letters of the remaining sequences. Set $s := sa$ and delete the front a from these sequences. Repeat this step until no sequences are left.
4. Output s.

It is shown in [8] that, on an alphabet of size k, Majority-Merge produces a common supersequence of length $O(n \log n)$ in the worst case and a common supersequence of length $(k+1)n/2 + O(\sqrt{n})$ on the average. In the next theorem, we will show that its average-case performance is actually near optimal.

Theorem 4.5. *For a set S containing n sequences over Σ of length n, the algorithm Majority-Merge approximates the SCS with an expected additive error $O(n^\delta)$, where $\delta = \frac{\sqrt{2}}{2} \approx 0.707$.*

Similar to the proof of Theorem 4.1, we first consider Kolmogorov random strings and obtain some tight upper bound on the performance of the algorithm Majority-Merge and lower bound on the length of the SCS. So, again, fix a Kolmogorov random string x of length n^2 and cut x into n equal length pieces $x_1, \ldots, x_n$. Let $S = \{x_1, ..., x_n\}$. It is shown in [8] that Majority-Merge actually finds a common supersequence of length $(k+1)n/2 + O(\sqrt{n})$ on the set S. Hence, it suffices to prove that $OPT(S) \geq (k+1)n/2 - O(n^\delta)$. We prove this in the following by using essential properties of Kolmogorov random strings.

Fix an arbitrary SCS $s = s_1 s_2 ... s_l$ for S and let $\mathcal{A}$ denote the procedure that produces s on set S by scanning the input sequences from left to right and merging common letters. We want to show that

$$l \geq (k+1)n/2 - O(n^\delta).$$

Note that (i) clearly $l \leq kn$ and (ii) $\mathcal{A}$ is uniquely determined by the SCS s. Let us arrange the input sequences $x_1, \ldots, x_n$ as an $n \times n$ matrix M with x_i being row i. For each $1 \leq i \leq l$, call the sequence, from top to bottom, consisting of the first letters in the remaining input sequences after $i-1$ steps of $\mathcal{A}$ the i-th *frontier.* Thus a frontier is just a jogged line from top to bottom, indicating which letter is being considered by $\mathcal{A}$ at the moment in each sequence x_i.

Since totally $\mathcal{A}$ merges n^2 letters in producing s, the number n^2/l represents the *average number* of letters merged by $\mathcal{A}$ in one step. We want to show that on the average, $\mathcal{A}$ merges at most $2n/(k+1) + O(n^\delta)$ letters in a step for some $\delta < 1$, *i.e.*,

$$n^2/l \leq 2n/(k+1) + O(n^\delta).$$

This would imply that $l \geq (k+1)n/2 - O(n^\delta)$.

CLAIM 4.6. *On the average, $\mathcal{A}$ merges at most $2n/(k+1) + O(n^\delta)$ letters in a step.*

Proof. Our basic idea is to show that the average merge amount takes its maximum when the supersequence s is of the form π^* for some permutation π of alphabet Σ, using the property that after each merge, the successors of the merged letters are "generated" according to a fair-coin rule, *i.e.*, the letters $a_1, \ldots, a_k$ must be distributed evenly among these "new" letters. This property holds because the matrix M is random.

For each $1 \leq i \leq l$ and $1 \leq j \leq k$, let $r_{i,j}$ denote the number of letter a_j contained in frontier i. Define $r_i = \sum_{j=1}^k r_{i,j}$ for each i to be the *length* of frontier i. Clearly $n = r_1 \geq \ldots \geq r_l$. Let l_0 be the smallest index such that $r_{l_0} \leq 2n/(k+1)$. Then we only need to prove an upper bound of $2n/(k+1) + O(n^\delta)$ on the average amount of merges made by $\mathcal{A}$ up to step l_0, since it merges at most $2n/(k+1)$ letters every step after step l_0. For each $1 \leq i \leq l_0$, denote the number of letters merged at step i as m_i.

We first would like to show that, at any step, if a large number of some letter a are merged, then the *new* letters immediately behind these a's in the involved input sequences should have approximately equal share of the letters $a_1, \ldots, a_k$. In view of Lemma 4.2, this is true as long as we can show that the subsequence consisting of these new letters in the next frontier is more or less random. We need the following lemma which indirectly proves the randomness of this subsequence.

LEMMA 4.7. *Let $frontier(i)$ be the list of positions indicating where the i-th frontier cuts through sequences $x_1, \ldots, x_n$. Let M' be the all the letters on the left of the i-th frontier including the i-th frontier, plainly listed column by column from left to right. Then,*

$$K(frontier(i)|s, i, M', n) \leq O(1).$$

Proof. Given s, i, M', n, we can simulate $\mathcal{A}$, together with s, on partial input M' for $i-1$ steps. Then we should have the positions for the i-th frontier. Note that in the listing of M', it is not necessary that each column is of length n, since some input sequences may have already run out. But this can be detected easily using n. Thus in all the future steps, we know that these sequences are not present any more, and can correctly arrange the letters of M'. □

In the following calculation, let ϵ be the solution of the following equation,

$$1 - 2\epsilon\delta = \delta$$

where $\delta = \frac{1}{2} + \epsilon$. So $\epsilon = (\sqrt{2}-1)/2 \approx 0.207$ and $\delta = \sqrt{2}/2 \approx 0.707$.

Let $i < l_0$ be any index. Suppose that the letter $a_j \in \Sigma$ is chosen to be merged by $\mathcal{A}$ at step i. If the letter a_j appears in the i-th frontier $\Omega(n^\delta)$ times, *i.e.*, $r_{i,j} = \Omega(n^\delta)$, then, by Lemma 4.2, if the letters $a_1, \ldots, a_k$ are unevenly distributed among the subsequence of frontier $i+1$ consisting of the successors of these merged a_j's, then we can compress this subsequence by $\Omega(n^{2\epsilon\delta})$ letters.

LEMMA 4.8. *There are at most $O(n^{1-2\epsilon\delta}) = O(n^\delta)$ subsequences of length $n^d = \Omega(n^\delta)$ such that some letter appears $n^{\frac{d}{2}+\epsilon}$ more often (or less often) than the average n^d/k in these subsequences.*

Proof. Otherwise we can describe the random matrix M by simply listing all the letters other than those appearing in the subsequences mentioned above, recording the supersequence s and the locations (*i.e.*, indices) of the subsequences, and compressing the subsequences using s. Since we save $\Omega(n^{2\epsilon\delta})$ letters on each such subsequence by Lemma 4.2, totally we save more than $\Omega(n^{1-2\epsilon\delta}) \cdot \Omega(n^{2\epsilon\delta}) = \Omega(n)$ letters. Thus we can encode x in less than $|x| - \log|x|$ letters. Notice that by Lemma 4.7, the position of the letters of frontier i in their corresponding input sequence can be derived from s, i, n, and the preceding frontiers. □

Let the frontiers containing the $q = O(n^\delta)$ unevenly distributed subsequences be indexed $p_1, \ldots, p_q$. Let $p_0 = 1$ and $p_{q+1} = l_0 + 1$. Cut M into $q+1$ sections: $M_0, \ldots, M_q$, where M_i begins at frontier p_i and ends at frontier $p_{i+1} - 1$.

Now we fix a section M_g and calculate the total amount of merges made by $\mathcal{A}$ in M_g. Since the letters in each set of new letters (after a merge) are distributed uniformly within this section, we have the following relation between $r_{i+1,j}$ and $r_{i,j}$. Suppose that the letter a_h is merged at step i, $p_g \le i < p_{g+1}$. Then $r_{i+1,j} = r_{i,j} + r_{i,h}/k \pm O(n^\delta)$, for each $j \neq h$, and $r_{i+1,h} = r_{i,h}/k \pm O(n^\delta)$. (Note that, the recurrence is automatically true if $r_{i,h} < O(n^\delta)$.) To simplify the presentation, below we will drop the minor term $O(n^\delta)$ when using the above recurrence relation, and simply add $O(n^\delta) \cdot (p_{g+1} - p_g)$ later on to the total amount of merges in section M_g. It is easy to verify that in the worst case this fluctuation of magnitude $O(n^\delta)$ can add at most $k \cdot O(n^\delta) = O(n^\delta)$ to the average merge amount.

Define a function $\rho(t_1, \ldots, t_k, i, j)$ satisfying the following recurrence relation:

$$\begin{aligned}
\rho(t_1, \ldots, t_k, i, 1) &= t_i, \quad 1 \le i \le k \\
\rho(t_1, \ldots, t_k, j \bmod k, j+1) &= \frac{\rho(t_1, \ldots, t_k, j \bmod k, j)}{k} \\
\rho(t_1, \ldots, t_k, j+1 \bmod k, j+1) &= \rho(t_1, \ldots, t_k, j+1 \bmod k, j) \\
&\quad + \frac{\rho(t_1, \ldots, t_k, j \bmod k, j)}{k} \\
&\cdots
\end{aligned}$$

$$\begin{aligned}\rho(t_1,\dots,t_k,j+k-1 \bmod k, j+1) &= \rho(t_1,\dots,t_k,j+k-1 \bmod k, j)\\ &\quad+\frac{\rho(t_1,\dots,t_k,j \bmod k,j)}{k}\end{aligned}$$

Intuitively, the function $\rho(t_1,\dots,t_k,i,j)$ corresponds to the distribution of letters $a_1,\dots,a_k$ in the frontiers if the letters are merged following the sequence π^*, where $\pi = a_{i_1}\cdots a_{i_k}$ is some permutation of Σ, and initially the letters $a_{i_1},\dots,a_{i_k}$ are distributed as $(t_1,\dots,t_k)$. Let

$$\mu(t_1,\dots,t_k,i) = \sum_{j=1}^{i} \rho(t_1,\dots,t_k,j \bmod k, j)$$

Thus $\mu(t_1,\dots,t_k,i)$ represents the total amount of merges achieved following the sequence π^* for i steps and given the initial distribution $(t_1,\dots,t_k)$ of letters $a_{i_1},\dots,a_{i_k}$. It can be shown that

$$\mu(t_1,\dots,t_k,i) = \sum_{j=1}^{k} \omega(i-j+1)\cdot t_j$$

where $\omega(i)$ is a function defined recursively below:

$$\begin{aligned}\omega(i) &= 0, \quad i \le 0\\ \omega(i) &= 1+\sum_{j=1}^{k}\omega(i-j)/k, \quad i\ge 1\end{aligned}$$

Intuitively, the number $\omega(i-j+1)$ represents the total contribution towards the amount of merges achieved in i steps from a letter that is merged at step j, $1\le j\le i$. Note that, $\omega(i)$ is an increasing function. We can prove the following lemma by induction.

LEMMA 4.9. *Let i be any index such that $p_g \le i \le p_{g+1}-1$. Suppose that $r_{i,j_1} \ge \cdots \ge r_{i,j_k}$, where $j_1,\dots,j_k$ is some permutation of $1,\dots,k$. Then $\sum_{j=i}^{p_{g+1}-1} m_j \le \mu(r_{i,j_1},\dots,r_{i,j_k},p_{g+1}-i)$.*

Proof. The lemma holds clearly if $i = p_{g+1}-1$. Now we prove that the lemma holds for i, assuming that it holds for $i+1$. For convenience, here we assume that $j_1 = 1,\dots,j_k = k$. Suppose that $\mathcal{A}$ merges the letter a_h at step i. Then

$$\begin{aligned}\sum_{j=i}^{p_{g+1}-1} m_j &= r_{i,h} + \sum_{j=i+1}^{p_{g+1}-1} m_j\\ &\le r_{i,h} + \mu(r_{i+1,1},\dots,r_{i+1,h-1},r_{i+1,h+1},\dots,r_{i+1,k},r_{i+1,h},p_{g+1}-i-1)\end{aligned}$$

where $r_{i+1,j} = r_{i,j} + r_{i,h}/k$, for each $j \ne h$, and $r_{i+1,h} = r_{i,h}/k$. Observe that that $r_{i+1,1} \ge \cdots \ge r_{i+1,h-1} \ge r_{i+1,h+1} \ge \cdots \ge r_{i+1,k} \ge r_{i+1,h}$. But

$$\begin{aligned}&r_{i,h} + \mu(r_{i+1,1},\dots,r_{i+1,h-1},r_{i+1,h+1},\dots,r_{i+1,k},r_{i+1,h},p_{g+1}-i-1)\\ =\; &\mu(r_{i,h},r_{i,1},\dots,r_{i,h-1},r_{i,h+1},\dots,r_{i,k},p_{g+1}-i)\end{aligned}$$

$$\begin{aligned}
&= \omega(p_{g+1}-i)\cdot r_{i,h} + \omega(p_{g+1}-i-1)\cdot r_{i,1} + \ldots + \omega(p_{g+1}-i-h+1)\cdot r_{i,h-1} \\
&\quad + \omega(p_{g+1}-i-h)\cdot r_{i,h+1} + \ldots + \omega(p_{g+1}-i-k+1)\cdot r_{i,k} \\
&\le \omega(p_{g+1}-i)\cdot r_{i,1} + \ldots + \omega(p_{g+1}-i-k+1)\cdot r_{i,k} \\
&= \mu(r_{i,1},\ldots,r_{i,k},p_{g+1}-i)
\end{aligned}$$

The above inequality holds because $\omega(j)$ is increasing and $r_{i,1} \geq \ldots \geq r_{i,k}$. □
Hence, the total amount of merges made by $\mathcal{A}$ in section M_g is

$$\begin{aligned}
\sum_{j=p_g}^{p_{g+1}-1} m_j &\le \mu(r_{p_g,1},\ldots,r_{p_g,k},p_{g+1}-p_g) \\
&= \sum_{j=1}^{k} \omega(p_{g+1}-p_g+1-j)\cdot r_{p_g,j} \\
&\le \sum_{j=1}^{k} \omega(p_{g+1}-p_g)\cdot r_{p_g,j} \\
&= \omega(p_{g+1}-p_g)\cdot \sum_{j=1}^{k} r_{p_g,j} \\
&\le \omega(p_{g+1}-p_g)\cdot n
\end{aligned}$$

We need one more lemma.

LEMMA 4.10. $\omega(i) \leq \frac{2i}{k+1} + 1$.

Proof. Clearly the lemma holds for all $i \leq 1$. Now suppose that it holds for all $i \leq h$ for some h. Then

$$\begin{aligned}
\omega(h+1) &= 1 + \sum_{j=1}^{k} \omega(h+1-j)/k \\
&\le 2 + \sum_{j=1}^{k} \frac{2(h+1-j)}{k(k+1)} \\
&= 2 + \frac{2(h+1)-k-1}{k+1} \\
&= 1 + \frac{2(h+1)}{k+1}
\end{aligned}$$

□ Therefore,

$$\sum_{j=p_g}^{p_{g+1}-1} m_j \le \omega(p_{g+1}-p_g)\cdot n \le \frac{2n(p_{g+1}-p_g)}{k+1} + n$$

Now we add $O(n^\delta)\cdot(p_{g+1}-p_g)$ back to the above bound and conclude that the total amount of merges in section M_g is at most

$$\frac{2n(p_{g+1}-p_g)}{k+1} + n + O(n^\delta)\cdot(p_{g+1}-p_g)$$

Thus, the total amount of merges in all sections is

$$
\begin{aligned}
& \sum_{g=0}^{q} \frac{2n(p_{g+1}-p_g)}{k+1} + n + O(n^\delta)\cdot(p_{g+1}-p_g) \\
= \;& \frac{2nl_0}{k+1} + n(q+1) + O(n^\delta)\cdot l_0 \\
= \;& \frac{2nl_0}{k+1} + n\cdot O(n^\delta) + O(n^\delta)\cdot l_0 \\
= \;& \frac{2nl_0}{k+1} + O(n^\delta)\cdot l_0
\end{aligned}
$$

That is, the overall average amount of merges is $\frac{2n}{k+1}+O(n^\delta)=\frac{2n}{k+1}+O(n^{0.707})$. This completes the proof of the claim. □

Proof of Theorem 4.5. Similar to the argument at the end of the proof for Theorem 4.1 and also because Majority-Merge produces a common supersequence of length $O(n\log n)$ in the worst case [8], the algorithm has an expected additive error $O(n^{0.707})$.

The above proof implies the following interesting corollary.

COROLLARY 4.11. *The expected length of an SCS for a set of n random sequences of length n is $(k+1)n/2 \pm O(n^{0.707})$.*

As we mentioned before, both Theorems 4.1 and 4.5 actually hold for inputs consisting of $p(n)$ sequences of length n, where $p()$ is some fixed polynomial.

5. Some Remarks. A problem that is closely related to the SCS problem is the shortest common superstring problem. Although this problem is also **NP**-hard, the status of its approximation complexity is quite different. Blum *et al.* have shown that shortest common superstring problem can be approximated within a factor of 3 in polynomial time [4]. They also showed that the problem (on an unbounded alphabet) is **MAX SNP**-hard.

Acknowledgments. We thank the referees for many constructive criticisms and suggestions, and C. Fraser for a correction.

REFERENCES

[1] A. Aho, J. Hopcroft, and J. Ullman. *Data Structures and Algorithms.* Addison-Wesley, Reading, Mass., 1983.

[2] K. Alexander. The rate of convergence of the mean length of the longest common subsequence. Manuscript, 1992.

[3] A. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof Verification and Hardness of Approximation Problems. *Proc. 33rd IEEE Symp. Found. Comp. Sci.*, 1992, 14–23.

[4] A. Blum, T. Jiang, M. Li, J. Tromp, and M. Yannakakis. Linear approximation of shortest superstrings. *Proc. 23rd ACM Symp. on Theory of Computing*, 1991, 328–336. Also to appear in *J. ACM.*

[5] V. Chvatal and D. Sankoff. Longest common subsequences of two random sequences. *J. Appl. Probab.* 12, 306–315.

[6] M.O. Dayhoff. Computer analysis of protein evolution. *Scientific American* 221:1(July, 1969), 86–95.

[7] D.E. Foulser. On random strings and sequence comparisons. *Ph.D. Thesis*, Stanford University, 1986.

[8] D.E. Foulser, M. Li, and Q. Yang. Theory and algorithms for plan merging. *Artificial Intelligence Journal* 57, 1992, 143–181.

[9] Garey and D. Johnson. *Computers and Intractability.* Freeman, New York, 1979.

[10] C.C. Hayes. A model of planning for plan efficiency: Taking advantage of operator overlap. *Proceedings of the 11th International Joint Conference of Artificial Intelligence*, Detroit, Michigan. (1989), 949–953.

[11] D.S. Hirschberg. The longest common subsequence problem. *Ph.D. Thesis*, Princeton University, 1975.

[12] W.J. Hsu and M.W. Du. Computing a longest common subsequence for a set of strings. *BIT* 24, 1984, 45–59.

[13] R.W. Irving and C.B. Fraser. Two algorithms for the longest common subsequence of three (or more) strings. *Proc. Symp. on Combinatorial Pattern Matching*, Tucson, Arizona, USA, May 1992.

[14] D. Karger, R. Motwani, and G.D.S. Ramkumar. On approximating the longest path in a graph. *Proc. Workshop on Algorithms and Data Structure*, Montreal, Canada, 421–432, 1993.

[15] R. Karinthi, D.S. Nau, and Q. Yang. Handling feature interactions in process planning. Department of Computer Science, University of Maryland, College Park, MD. (1990).

[16] M. Li and P.M.B. Vitányi. Kolmogorov complexity and its applications. *Handbook of Theoretical Computer Science, Vol. A*, 187–254, J. van Leeuwen, Ed., Elsevier/MIT Press, 1990.

[17] M. Li and P.M.B. Vitányi. *An introduction to Kolmogorov complexity and its applications.* Springer-Verlag, 1993.

[18] M. Li and P.M.B. Vitányi. Combinatorial properties of finite sequences with high Kolmogorov complexity. To appear in *Math. Syst. Theory.*

[19] S.Y. Lu and K.S. Fu. A sentence-to-sentence clustering procedure for pattern analysis. IEEE Trans. Syst., Man, Cybern. Vol. SMC-8(5), 1978, 381–389.

[20] C. Lund and M. Yannakakis. On the hardness of approximating minimization problems. Proc. ACM Symp. Theory of Computing, 1993, 286–293.

[21] D. Maier. The complexity of some problems on subsequences and supersequences. *J. ACM*, 25(1978), 322–336.

[22] C.H. Papadimitriou and M. Yannakakis. Optimization, Approximation, and Complexity Classes. Extended abstract in *Proc. 20th ACM Symp. on Theory of Computing.* 1988, 229–234; full version in *Journal of Computer and System Sciences* 43, 1991, 425–440.

[23] K. Raiha and E. Ukkonen. The shortest common supersequence problem over binary alphabet is NP-complete. *Theoretical Computer Science* 16(1981), 187–198.

[24] D. Sankoff and J. Kruskall (Eds.) *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison.* Addison-Wesley, Reading, MA, 1983.

[25] T. Sellis. Multiple query optimization. *ACM Transactions on Database Systems*, 13:1(1988), 23–52.

[26] T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. Journal of Molecular Biology, 147(1981), 195–197.

[27] J. Storer. *Data compression: methods and theory.* Computer Science Press, 1988.

[28] V.G. Timkovskii. Complexity of common subsequence and supersequence problems and related problems. *English Translation from Kibernetika*, 5(1989), 1–13.

[29] R.A. Wagner and M.J. Fischer. The string-to-string correction problem. J. ACM, 21:1(1974), 168–173.