INFORMATION AND COMPUTATION **124**, 145–153 (1996)
ARTICLE NO. 0011

Maximal Common Subsequences and Minimal Common Supersequences*

CAMPBELL B. FRASER[†] AND ROBERT W. IRVING

Computing Science Department, University of Glasgow, Glasgow G12 8QQ, Scotland

AND

MARTIN MIDDENDORF

Institut für Angewandte Informatik und Formale Beschreibungsverfahren, D-76128 Universität Karlsruhe, Germany

The problems of finding a longest common subsequence and a shortest common supersequence of a set of strings are well known. They can be solved in polynomial time for two strings (in fact the problems are dual in this case), or for any fixed number of strings, by dynamic programming. But both problems are NP-hard in general for an arbitrary number *k* of strings. Here we study the related problems of finding a shortest maximal common subsequence and a longest minimal common supersequence. We describe dynamic programming algorithms for the case of two strings (for which case the problems are no longer dual), which can be extended to any fixed number of strings. We also show that both problems are NP-hard in general for *k* strings, although the latter problem, unlike shortest common supersequence, is solvable in polynomial time for strings of length 2. Finally, we prove a strong negative approximability result for the shortest maximal common subsequence problem. © 1996 Academic Press, Inc.

1. INTRODUCTION

A *subsequence* of a string α is any string that can be obtained from α by the deletion of zero or more symbols. A *supersequence* of α is any string that can be obtained from α by the insertion of zero or more symbols. Given a set S of k strings, a *common subsequence* of S is a string that is a subsequence of every string in S and a *common supersequence* of S is a string that is a supersequence of every string in S.

The *longest common subsequence* (LCS) and *shortest common supersequence* (SCS) problems are classical problems of stringology, with important applications in computational biology, file comparison, data compression, etc.

A common subsequence α is *maximal* if no proper supersequence of α is also a common subsequence of S—in other words, if α is not contained as a subsequence in any longer common subsequence of S. A *shortest maximal common subsequence* (*smcs*) of S is a maximal common subsequence of shortest possible length. Clearly, a maximal common subsequence of *greatest* possible length is just a longest common subsequence, a concept that has been widely explored in the literature.

A common supersequence α is *minimal* if no proper subsequence of α is also a common supersequence of S—in other words, if α does not contain as a subsequence any shorter common supersequence of S. A *longest minimal common supersequence* (*lmcs*) of S is a minimal common supersequence of longest possible length. Clearly, a minimal common supersequence of *shortest* possible length is just a shortest common supersequence.

EXAMPLE. If $\alpha_1 = abc$, $\alpha_2 = bca$, the maximal common subsequences are a, bc, and the unique smcs is a, of length 1.

The minimal common supersequences are *bcabc*, *abca*, *bacbac*, and the unique lmcs is *bacbac*, of length 6.

In this paper, we study the shortest maximal common subsequence problem (SMCS) and and the longest minimal common supersequence problem (LMCS) from the complexity point of view. The problems are of genuine interest in their own right, although the original motivation for the study of maximal common subsequences and minimal common supersequences was in the context of approximation algorithms for LCS and SCS. For instance, any approximation algorithm for the LCS of a set of strings will return a common subsequence of the strings, which can then be made maximal by the insertion of zero or more additional characters. The question arises as to the extent to which the length of such a maximal common subsequence might differ from the length of a longest common subsequence. A similar situation holds in the case of approximation algorithms for the SCS of a set of strings.

We show that, like the LCS and SCS problems, both of these new problems can be solved in polynomial time by

* A preliminary version of this paper, by the first two authors, was presented at CPM'94, the fifth International Conference on Combinatorial Pattern Matching, at Asilomar, California, in June 1994.

[†] Supported by a postgraduate research studentship from the Engineering and Physical Sciences Research Council.

dynamic programming for $k=2$ (and, by extending the algorithms, for any fixed value of k). However, the dynamic programming algorithms are not quite so straightforward as those for the LCS and SCS problems, and they have complexities $O(m^2n)$ and $O(mn(m+n))$, respectively, for strings of lengths m and n. Note that the existence of polynomial-time algorithms for the SMCS and LMCS problems in the case of two strings is by no means obvious. Consider the problem of finding a maximum cardinality matching in a bipartite graph. This problem is well known to be solvable in polynomial time, whereas the problem of finding a minimum maximal bipartite matching is NP-hard [YG80].

We also show that, as is the case for the LCS and SCS problems, SMCS and LMCS are NP-hard when the number of strings k becomes a problem parameter. However, we pinpoint one interesting difference between the SCS and LMCS problems, namely that the latter, unlike the former, is solvable in polynomial time for strings of length 2. Furthermore, we prove a strong negative result regarding the likely existence of good polynomial-time approximation algorithms for SMCS in the case of general k. We leave open the possibility of a good polynomial-time approximation algorithm for LMCS.

2. THE SMCS PROBLEM FOR TWO STRINGS

When restricted to the case of just two strings α and β of lengths m and n respectively, the classical LCS and SCS problems are easily solvable in $O(mn)$ time by dynamic programming. Indeed, in this case, the LCS and SCS problems are dual, in that $s=m+n-l$, where l and s are the lengths of a longest common subsequence and shortest common supersequence respectively. Much effort has gone in to finding refinements of dynamic programming and other approaches which lead to improvements in complexity in many cases. See, for example, [AG87, Hir75, Hir77, HS77, Ukk85, WMMM90].

The following simple example serves to illustrate the fact that there is no obvious corresponding duality between the SMCS and LMCS problems in the case of two strings.

Example. Let $\alpha = abc$, $\beta = dab$. Then the only maximal common subsequence is ab of length 2, while $dabc$, $abcdab$, and $abdcab$ are some of the minimal common supersequences, the latter two being lmcs's.

It is true, however, that if γ is a maximal common subsequence of length r of α and β, then forming an alignment of α and β in which the elements of γ are matched reveals a minimal common supersequence, δ say, of α and β of length $m+n-r$. For if δ were not minimal, then some single symbol c could be removed from δ to give a shorter common supersequence δ'. But the symbol of α or β (or both) represented by c in the alignment must be matched with new symbols in the alignment corresponding to δ', leading to a contradiction of the maximality of γ. Hence, if l' is the length of an smcs, and s' the length of an lmcs, it follows that $s' \geqslant m+n-l'$. That the inequality can be strict is shown by the above example.

Hence the question arises as to whether either or both of the SMCS and LMCS problems can be solved in polynomial time, by dynamic programming or otherwise.

In this section we describe a polynomial-time algorithm to determine the length of an smcs of two strings, and in the following section a polynomial-time algorithm to determine the length of an lmcs of two strings. It turns out that these algorithms will determine the lengths of all maximal common subsequences and all minimal common supersequences respectively. They will also allow the construction of an smcs, and indeed of all the maximal common subsequences (respectively an lmcs, and all minimal common supersequences) of the two strings.

The algorithms use a dynamic programming approach based, as usual, on a table that relates the ith prefix $\alpha^i = \alpha[1 \ldots i]$ of α and the jth prefix $\beta^j = \beta[1 \ldots j]$ of β, for $i = 1, \ldots, m$, $j = 1, \ldots, n$, where m, n are the lengths of α, β respectively. However, as we shall see, for each i, j we need retain rather more information than merely the lengths of the maximal common subsequences, or of the minimal common supersequences, of α^i and β^j.

The SMCS Algorithm. Given a string α and a subsequence γ of α, we define

$sp(\alpha, \gamma)$ = length of the shortest prefix of α that is a supersequence of γ

Given strings α, β of lengths m and n respectively, we define the set S_{ij}, for each $i = 1, \ldots, m$, $j = 1, \ldots, n$, by

$S_{ij} = \{(r, (x, y))$: α^i and β^j have a maximal common subsequence γ of length r, and $sp(\alpha, \gamma) = x$, $sp(\beta, \gamma) = y\}$

with

$$S_{00} = \{(0, (0, 0))\}$$

For string α of length m, position i, and symbol a, we define

$$next_\alpha(i, a) = \begin{cases} \min\{k : \alpha[k] = a, k > i\} & \text{if such a } k \text{ exists} \\ m+1 & \text{otherwise.} \end{cases}$$

If α is a string and a a symbol of the alphabet, we denote by $\alpha + a$ the string obtained by appending a to α. Likewise, if the last character of α is a, we denote by $\alpha - a$ the string obtained by deleting the final a from α.

The algorithm for SMCS is based on a dynamic programming scheme for the sets S_{ij} defined above. So evaluation of S_{mn} reveals the length of an smcs, and also finds the lengths of all maximal common subsequences of α and β (indeed of all maximal common subsequences of all pairs of prefixes of α and β). Furthermore, the use of suitable tracebacks in the array of S_{ij} values, can be used to generate, not only an smcs, but all maximal common subsequences.

The basis of the dynamic programming scheme is contained in the following theorem:

THEOREM 1. (i) *If* $\alpha[i] = \beta[j] = a$ *then*

$$S_{ij} = \{(r, (next_\alpha(x, a), next_\beta(y, a))): (r-1, (x, y)) \in S_{i-1, j-1}\}$$

(ii) *If* $\alpha[i] \neq \beta[j]$ *then*

$$S_{ij} = \{(r, (x, j)) \in S_{i-1, j}\} \cup \{(r, (i, y)) \in S_{i, j-1}\} \cup (S_{i-1, j} \cap S_{i, j-1}).$$

Proof (i) Suppose $(r-1, (x, y)) \in S_{i-1, j-1}$ and that γ' is a maximal common subsequence of α^{i-1} and β^{j-1} of length $r-1$ with $sp(\alpha, \gamma') = x$ and $sp(\beta, \gamma') = y$. Then it is immediate that $\gamma = \gamma' + a$ is a maximal common subsequence of α^i and β^j, and that $sp(\alpha, \gamma) = next_\alpha(x, a)$, $sp(\beta, \gamma) = next_\beta(y, a)$.

On the other hand, suppose that γ is a maximal common subsequence of length r of α^i and β^j. Then the last symbol of γ is a, and $\gamma' = \gamma - a$ is certainly a common subsequence of α^{i-1} and β^{j-1}. If it were not maximal, then some supersequence δ of γ' would be a common subsequence of α^{i-1} and β^{j-1}, and therefore $\delta + a$, a supersequence of γ, would be a common subsequence of α^i and β^j, contradicting the maximality of γ. So $(r-1, (sp(\alpha, \gamma'), sp(\beta, \gamma')) \in S_{i-1, j-1}$ and $(r, (sp(\alpha, \gamma), sp(\beta, \gamma)) \in S_{ij}$ with $sp(\alpha, \gamma) = next_\alpha(sp(\alpha, \gamma'), a)$ and $sp(\beta, \gamma) = next_\beta(sp(\beta, \gamma'), a)$.

(ii) Suppose $(r, (x, j)) \in S_{i-1, j}$, and that γ is a maximal common subsequence of α^{i-1} and β^j of length r with $sp(\alpha, \gamma) = x$, $sp(\beta, \gamma) = j$. Then γ is a common subsequence of α^i and β^j, and must be maximal since $\gamma + \alpha[i]$ cannot be a subsequence of β^j. A similar argument holds for $(r, (i, y)) \in S_{i, j-1}$. So $\{(r, (x, j)) \in S_{i-1, j}\} \cup \{(r, (i, y)) \in S_{i, j-1}\} \subseteq S_{ij}$.

Further, if $(r, (x, y)) \in S_{i-1, j} \cap S_{i, j-1}$, then there is a string γ with $sp(\alpha, \gamma) = x < i$, $sp(\beta, \gamma) = y < j$, of length r, which is a maximal common subsequence of α^i and β^{j-1}, and of α^{i-1} and β^j. So γ must also be a maximal common subsequence of α^i and β^j. For any supersequence of γ that is a subsequence of α^i and β^j must either be a subsequence of α^i and β^{j-1}, or of α^{i-1} and β^j.

On the other hand, suppose that γ is a maximal common subsequence of length r of α^i and β^j.

Case (iia). $sp(\alpha, \gamma) = i$. Then γ is a maximal common subsequence of α^i and β^{j-1}, and so $(r, (i, y)) \in S_{i, j-1}$ for some y.

Case (iib). $sp(\beta, \gamma) = j$. Then γ is a maximal common subsequence of α^{i-1} and β^j, and so $(r, (x, j)) \in S_{i-1, j}$ for some x.

Case (iic). $sp(\alpha, \gamma) < i$, $sp(\beta, \gamma) < j$. Then γ is a maximal common subsequence both of α^{i-1} and β^j, and of α^i and β^{j-1}. So $(r, (sp(\alpha, \gamma), sp(\beta, \gamma))) \in S_{i-1, j} \cap S_{i, j-1}$.

This completes the proof of the theorem. ▮

Recovering a Shortest Maximal Common Subsequence. The recovery of a particular smcs involves a standard type of traceback through the dynamic programming table from cell (m, n), during which the sequence is constructed in reverse order. To facilitate this traceback, each entry in position (i, j) in the table (for all i, j) should have associated with it, during the application of the dynamic programming scheme, one or more pointers indicating which particular element(s) in cells $(i-1, j)$, $(i, j-1)$, or $(i-1, j-1)$ led to the inclusion of that element in cell (i, j). For example, if $\alpha[i] = \beta[j] = a$, and $(r-1, (x, y)) \in S_{i-1, j-1}$ then $(r, (next_\alpha(x, a), next_\beta(y, a))$ is placed in cell (i, j) with a pointer to the element $(r-1, (x, y))$ in cell $(i-1, j-1)$.

With these pointers, any path from an element $(r, (x, y))$ in cell (m, n) to the element in cell (0,0) represents a maximal common subsequence of α and β of length r, namely the reversed sequence of matching symbols from the two strings corresponding to cells from which the path takes a diagonal step.

Analysis of the SMCS Algorithm. The number of cells in the dynamic programming table is essentially mn, so that if we could show that the number of entries in each cell was bounded by, say, $\min(m, n)$, and that the total amount of computation was bounded by a constant times the total number of table entries, then we would have a cubic time worst-case bound for the complexity of the algorithm. However, this turns out not to be the case, as the following example shows.

EXAMPLE. Consider two strings of length $n = p(p+1)/2 + q$ over an alphabet $\Sigma = \{a_1, ..., a_n\}$, defined as

$$\alpha = \alpha_1 + \alpha_2 + \cdots + \alpha_p + a_{p(p+1)/2+1} \ldots a_n$$

$$\beta = \alpha_p + \alpha_{p-1} + \cdots + \alpha_1 + a_n \ldots a_{p(p+1)/2+1},$$

where $\alpha_1 = a_1$, $\alpha_2 = a_2 a_3, ..., \alpha_p = a_{(p-1)p/2+1} \ldots a_{p(p+1)/2}$, and $+$ denotes concatenation.

In the dynamic programming table for strings α and β, position (n, n) contains the pq entries $(r, (x, y))$ for

$r=2, ..., p+1$, $x=p(p+1)/2+1, ..., n$, $y=n+1+p(p+1)/2-x$. Here, entry $(r, (x, y))$ arises from the maximal common subsequence $a_{t+1} \dots a_{t+r-1} a_x$, where $t=(r-2)(r-1)/2$. With $q=\Theta(p^2)$, this gives $\Theta(n^{3/2})$ entries in the (n, n)th cell.

However, suppose that we wish to find only the length of an smcs (and to construct such a sequence by traceback through the table). Then, if any particular cell in the table contains more then one entry $(r, (x, y))$ with the same (x, y) component, we may discard all but the one with the smallest r value. For if a maximal common subsequence γ has a prefix γ' such that $sp(\alpha, \gamma')=x$ and $sp(\beta, \gamma')=y$ then to make γ as short as possible, γ' must be chosen as short as possible.

Also, if the entries $(r, (x, y))$ in the (i, j)th cell are listed in increasing order of x, then clearly they must also be in decreasing order of y, and therefore, since $x \leqslant i, y \leqslant j$, the number of such entries with distinct (x, y) components cannot exceed $\min(i, j)$. Further, it is easy to see that by processing the lists of cell entries in this fixed order, the amount of work done in computing the contents of cell (i, j) is, bounded in case (i) by a constant times the number of entries in cell $(i-1, j-1)$, and in case (ii) by a constant times the sum of the numbers of entries in cells $(i-1, j)$ and $(i, j-1)$. (In case (i), this assumes precomputation of the tables of next values, which can easily be achieved in $O(n|\Sigma|)$ time for a string of length n, where Σ is the alphabet.)

In conclusion, the length of an smcs can be established by a suitably amended version of the above dynamic programing scheme in $O(m^2 n)$ time in the worst case, for strings of lengths m and $n (m \leqslant n)$. Furthermore, such a subsequence can also be constructed from the dynamic programming table without increasing that overall time bound. But it remains open whether the lengths of all maximal common subsequences can be established within that time bound. A trivial bound of $O(m^3 n)$ applies in that case, since the number of entries in each cell is certainly bounded by m^2.

3. THE LMCS PROBLEM FOR TWO STRINGS

The LMCS algorithm is not dissimilar in spirit to the SMCS algorithm, and there is a certain duality involving the terms in which the algorithm is expressed.

Given strings α and γ, we define

$lp(\alpha, \gamma)$ = length of the longest prefix of α that is a subsequence of γ.

Given strings α, β of lengths m and n respectively, we define the set T_{ij} for each $i=0, ..., m, j=0, ..., n$ by

$T_{ij}=\{(r, (x, y))$: there exists a minimal common supersequence γ of α^i and β^j, of length r, such that $lp(\alpha, \gamma)=x, lp(\beta, \gamma)=y\}$

Finally, for string α, position i and symbol a, we define

$$f_\alpha(i, a)=\begin{cases} i+1 & \text{if } \alpha[i+1]=a \\ i & \text{otherwise.} \end{cases}$$

The algorithm for LMCS is based on a dynamic programming scheme for the sets T_{ij} defined above. So evaluation of T_{mn} reveals the length of an lmcs, but also finds the lengths of all minimal common supersequences of α and β (indeed of all minimal common supersequences of all pairs of prefixes of α and β). Furthermore, suitable tracebacks in the array of T_{ij} values can be used to generate, not only an lmcs, but all minimal common supersequences.

The 0th row and column of the T_{ij} table can be evaluated trivially, as follows:

$$T_{i0}=\{(i, (i, lp(\beta, \alpha^i)))\} \quad (1 \leqslant i \leqslant m)$$

and

$$T_{0j}=\{(i, (lp(\alpha, \beta^j), j))\} \quad (1 \leqslant j \leqslant n)$$

with

$$T_{00}=\{(0, (0, 0))\}.$$

The basis of the dynamic programming scheme is contained in the following theorem:

THEOREM 2. (i) *If* $\alpha[i]=\beta[j]=a$ *then*

$$T_{ij}=\{(r, (f_\alpha(x, a), f_\beta(y, a)) \colon (r-1, (x, y)) \in T_{i-1, j-1}\}.$$

(ii) *If* $\alpha[i]=a \neq b=\beta[j]$ *then*

$$T_{ij}=\{(r, (f_\alpha(x, b), j)) \colon (r-1, (x, j-1)) \in T_{i, j-1}\} \cup \{(r, (i, f_\beta(y, a)) \colon (r-1, (i-1, y)) \in T_{i-1, j}\}.$$

Proof

(i) Suppose $(r-1, (x, y)) \in T_{i-1, j-1}$ and that γ' is a minimal common supersequence of α^{i-1} and β^{j-1} of length $r-1$ with $lp(\alpha, \gamma')=x$ and $lp(\beta, \gamma')=y$. Then it is immediate that $\gamma=\gamma'+a$ is a minimal common supersequence, of length r, of α^i and β^j, and that $lp(\alpha, \gamma)=f_\alpha(x, a)$ and $lp(\beta, \gamma)=f_\beta(y, a)$.

On the other hand, suppose that γ is a minimal common supersequence of length r of α^i and β^j. Then $\gamma[r]=a$, and $\gamma'=\gamma-a$ is certainly a common supersequence of α^{i-1} and β^{j-1}. If γ' were not minimal, then some subsequence δ of γ' would be a common supersequence of α^{a-1} and β^{j-1}, and therefore $\delta+a$, a subsequence of γ, would be a common supersequence of α^i and β^j, contradicting the minimality or γ. So $(r-1, (x, y)) \in T_{i-1, j-1}$ with $x = lp(\alpha, \gamma')$, $y = lp(\beta, \gamma')$ and $lp(\alpha, \gamma) = f_\alpha(x, a)$, $lp(\beta, \gamma) = f_\beta(y, a)$.

(ii) Suppose $(r-1, (i-1, y)) \in T_{i-1, j}$, and that γ' is a minimal common supersequence of α^{i-1} and β^j of length $r-1$ with $lp(\alpha, \gamma') = i-1$, $lp(\beta, \gamma') = y$. (The argument is similar in the case $(r-1, (x, j-1)) \in T_{i, j-1}$.) Then $\gamma = \gamma' + a$ is a common supersequence of α^i and β^j with $lp(\alpha, \gamma) = i$ and $lp(\beta, \gamma) = f_\beta(y, a)$. Further, γ must be minimal. For suppose that a subsequence δ of γ is a common supersequence of α^i and β^j. If δ were a subsequence of γ', then γ' would not be a minimal common supersequence of α^{i-1} and β^j. So $\delta = \delta' + a$, where δ' is a subsequence of γ'. So δ' cannot be a common supersequence of α^{i-1} and β^j. If it is not a supersequence of α^{i-1} then $\delta' + a$ cannot be a supersequence of α^i—a contradiction. If it is not a supersequence of β^j then, since $\delta' + a$ is a supersequence of β^j, we must have $\beta[j] = a$—a contradiction.

On the other hand, suppose that γ is a minimal common supersequence of length r of α^i and β^j. Then $\gamma[r] = a$ or b.

Case (iia). $\gamma[r] = a$. It is immediate that, $lp(\alpha, \gamma) = i$, for otherwise $\gamma - a$ would be a common supersequence of α^i and β^j. So $\gamma' = \gamma - a$ is a minimal common supersequence of α^{i-1} and β^j with $lp(\alpha, \gamma') = i-1$ and $lp(\beta, \gamma') = y$ for some y such that $lp(\beta, \gamma) = f_\beta(y, a)$

Case (iib). $\gamma[r] = b$. A similar argument shows that $\gamma' = \gamma - b$ is a minimal common supersequence of α^i and β^{j-1} with $lp(\beta, \gamma') = j-1$ and $lp(\alpha, \gamma') = x$ for some x such that $lp(\alpha, \gamma) = f_\alpha(x, b)$.

This completes the proof of the theorem. ▮

Recovering a Longest Minimal Common Supersequence. As in the case of an smcs, the recovery of a particular lmcs involves a traceback through the dynamic programming table from cell (m, n) to cell $(0, 0)$, during which the sequence is constructed in reverse order. To facilitate the traceback, each entry in position (i, j) in the table (for all i, j) should have associated with it, during the application of the dynamic programming algorithm, one or more pointers indicating which particular element(s) in cells $(i-1, j)$, $(i, j-1)$, or $(i-1, j-1)$ led to the inlusion of that element in cell (i, j). For example, if $\alpha[i] = a = \beta[j]$ and $(r-1, (x, y)) \in T_{i-1, j-1}$, then $(r, (f_\alpha(x, a), f_\beta(y, a)))$ is placed in cell (i, j) with a pointer to the element $(r-1, (x, y))$ in cell $(i-1, j-1)$.

With these pointers, any path from an element $(r, (x, y))$ in cell (m, n) to the element in cell $(0, 0)$ represents a minimal common supersequence of α and β of length r, namely the reversed sequence of symbols found by recording $\alpha[i]$ for a vertical or diagonal step from cell (i, j) and $\beta[j]$ for a horizontal step from cell (i, j).

Analysis of the LMCS Algorithm. As in the case of the SMCS algorithm, we can establish a cubic time bound for the restricted version of the LMCS algorithm that is designed to find the length of an lmcs, and to construct such a common supersequence from the dynamic programming table. The trick again is the observation that, for this purpose, whenever $(r, (x, y))$ elements in the same cell have the same (x, y) component, only one need be retained namely that with the largest r value. For if a minimal common supersequence γ has a prefix γ' such that $lp(\alpha, \gamma') = x$ and $lp(\beta, \gamma') = y$, then to make γ as long as possible, γ' should be chosen as long as possible.

By this means we can restrict the number of elements in the (i, j)th cell to at most $i + j$, recalling that each such entry $(r, (x, y))$ has either $x = i$ or $y = j$. This leads to a worst-case time bound of $O(mn(m+n))$ for this version of the algorithm. Again, it is not clear whether the lengths of all minimal common supersequences can be found in time better than $O(mn(m+n)^2)$ in the worst case, this arising from the obvious upper bound of $(m+n)^2$ on the number of elements in each cell of the table.

4. THE SMCS PROBLEM FOR K STRINGS.

It was first proved by Maier [Mai78] that the problem of finding an LCS of k strings is NP-hard, even in the case of a binary alphabet. Further, Jiang and Li [JL94] showed that, unless P = NP, there cannot exist a polynomial-time approximation algorithm for LCS with a performance guarantee of k^δ, for some $\delta > 0$.

As we shall see in Theorem 3, the transformation, from Independent Set, given by Maier [Mai78] to prove the NP-completeness for LCS also serves as a transformation from the Minimum Independent Dominating Set problem to SMCS. The former problem is also NP-hard [GJ79], and was shown by Irving [Irv91] not to have a polynomial-time approximation algorithm with a constant performance guarantee (if P ≠ NP). Halldórsson [Hal93] strengthened this result to show that, if P ≠ NP, then for no $\delta < 1$ can there exist a polynomial-time approximation algorithm with performance guarantee k^δ. Maier's transformation has the property that the strings constructed have an LCS of length r if and only if the original graph has an independent set of size r. If the given graph has k *edges* then the derived LCS instance has $k+1$ strings. It will therefore follow from the transformation, not only that SMCS is NP-hard, but also that this same strongly negative approximability result applies to the SMCS problem.

THEOREM 3. (i) *The SMCS problem is NP-hard.*

(ii) *If P ≠ NP, then for no $\delta < 1$ can there exist a polynomial-time approximation algorithm for SMCS on k strings with performance guarantee k^δ.*

Proof. (i) Let $G = (V, E)$, t, with $V = \{v_1, ..., v_n\}$, and $E = \{e_1, e_2, ..., e_m\}$, be an arbitrary instance of (the decision version of) the Minimum Independent Dominating Set problem. We construct an instance of SMCS as follows. Include in the set S of strings the string $\alpha_0 = v_1 v_2 \dots v_n$. For each edge $e_i = \{v_p, v_q\}$ $(p < q)$ include in the set S of strings the string α_i defined by

$$\alpha_i = v_1 v_2 \dots v_{p-1} v_{p+1} \dots v_n v_1 v_2 \dots v_{q-1} v_{q+1} \dots v_n.$$

We claim that G has an independent dominating set of size t if and only if S has a maximal common subsequence of length t.

To prove this claim, we must show that (a) if G has an independent dominating set U of size t then S has a maximal common subsequence of length t; (b) if S has a maximal common subsequence of length t then G has an independent dominating set of size t.

To prove (a), assume that $U = \{v_{u_1}, v_{u_2}, \dots, v_{u_t}\}$ is an independent dominating set of size t, where $1 \leqslant u_1 < u_2 < \cdots < u_t \leqslant n$.

The string $\alpha = v_{u_1} v_{u_2} \dots v_{u_t}$ is clearly a subsequence of α_0 and of any α_i not formed from an edge connecting two vertices of U. Since U is an independent set, α is a common subsequence of S. If some supersequence, α', of α is a common subsequence of S then observe for a contradiction that $\exists v_p$ in α' but not in α, which is connected to $v_q \in U$ in G by edge $e_j = \{v_p, v_q\}$, since U is dominating. Assuming $p < q$, the string $\alpha_j = v_1 v_2 \dots v_{p-1} v_{p+1} \dots v_n v_1 v_2 \dots v_{q-1} v_{q+1} \dots v_n$. For α' to be a subsequence of α_0, v_p must precede v_q in α'. But this prevents α' from being a subsequence of α_j. A similar contradiction is obtained if $p > q$ is assumed.

To prove (b)), assume $\alpha = v_{u_1} v_{u_2} \dots v_{u_t}$, of length t, is a maximal common subsequence of the strings in S. The first observation is that if v_{u_p} and v_{u_q} are two symbols in α and $p < q$ then $u_p < u_q$. For otherwise α could not be a subsequence of α_0. The elements of α must form an independent set, U, of size t, in G. To see this, observe for a contradiction that if two elements, v_{u_p} and $v_{u_q} (p < q)$, of α are connected in G by edge $e_j = \{v_{u_p}, v_{u_q}\}$ then the string $v_{u_p} v_{u_q}$, a subsequence of α, would not be a subsequence of α_j (the string formed from e_j) and hence α would not be a subsequence of α_j. If U is not maximal then $\exists U'$ an independent set of size $t' > t$, and $U \subset U'$. This would imply there is some vertex v_j that is a member of U' but not a member of U. In that case, the string $\alpha' = v_{u_1} \dots v_{u_p} v_j v_{u_{p+1}} \dots v_{u_t}$, where $u_p < j < u_{p+1}$, a supersequence of α, would be a common subsequence of all the strings in S, contradicting the maximality of α. This concludes the proof of part (i).

The proof of part (ii) follows from the observation that the reduction is linear [PY91], and therefore preserves the aproximability of the Minimum Independent Dominating Set, and from the result of Halldórsson [Hal93] on the approximability of that problem. ∎

5. THE LMCS PROBLEM FOR *K* STRINGS

It is well known that the problem of finding a shortest common supersequence of k strings is NP-hard [Mai78], even in many restricted cases, such as

- over a binary alphabet [RU81], even if all strings have the same length and all contain precisely two 1's [Mid92];
- when all strings have length $\leqslant 3$ and each character appears $\leqslant 2$ times in total [Tim89];
- when all strings have length $\leqslant 2$ and each character appears $\leqslant 3$ times in total [Tim89].

We now show that the LMCS problem is also NP-hard in the general case. But in contrast to the SCS problem we can give a linear-time algorithm for LMCS if the strings are of length 2. The complexity of LMCS for strings of constant length > 2 is left open.

THEOREM 4. *The LMCS problem is NP-hard.*

Proof. Let positive integers $t, n = 3m$, and the set of positive integers $A = \{s_1, s_2, \dots, s_n\}$, with $\frac{1}{4}t < s_i < \frac{1}{2}t$ for each i, constitute an arbitrary instance of the 3-partition problem. This problem asks whether there exists a partition $\{A_1, A_2, \dots, A_m\}$ of A into sets of size 3 such that $\sum_{s_j \in A_i} s_j = t$ holds for all $i \in [1\colon m]$. The 3-partition problem is known to be NP-complete [GJ79].

Without loss of generality let $\sum_{i \in [1\colon n]} s_i = mt$ and $m > 3$. We construct an instance of LMCS as follows. For each $i \in [1\colon m-1]$ include in the set S of strings the string

$$\beta_i = b^{it} d b^{(m-i)t}.$$

For each $s_i \in A$ include in the set S the string α_i defined by

$$\alpha_i = u_i (b c_1 c_2 \dots c_r d)^{s_i - 1} b c_1 c_2 \dots c_r v_i,$$

where $r = 2mt$. We claim that A has a partition $\{A_1, A_2, \dots, A_m\}$ into sets of size 3 such that $\sum_{s_j \in A_i} s_j = t$ holds for all $i \in [1\colon m]$ iff S has a minimal common supersequence of length $t' = n + mt(r+2) + m - 1$.

Assume that $\{A_1, A_2, \dots, A_m\}$ is a partition of A into sets of size 3 such that $\sum_{s_j \in A_i} s_j = t$ holds for all $i \in [1\colon m]$. Without loss of generality assume that $A_i = \{s_{3i-2}, s_{3i-1}, s_{3i}\}$ for $i \in [1\colon m]$. Then it is easy to verify that the string $\alpha = \alpha_1 \alpha_2 \alpha_3 d \alpha_4 \alpha_5 \alpha_6 d \dots d \alpha_{n-2} \alpha_{n-1} \alpha_n$ is a minimal common supersequence of S of length $t' = n + mt(r+2) + m - 1$.

On the other hand assume that S has a minimal common supersequence of length t'. We need the following claim.

CLAIM 1. *Every minimal common supersequence β of $\{\beta_1, \beta_2, \dots, \beta_{m-1}\}$ has length $\leqslant 2(m-1)\,t + m - 1$.*

Proof of Claim 1. Clearly β contains at most $m-1$ d's. Since β is minimal, one of the following cases holds for each fixed occurrence of b in β: (1) $\beta=\delta_1 d\delta_2$, where the fixed occurrence of b is in δ_1 and δ_1 contains exactly it b's for an $i \in [1: m-1]$, or (2) $\beta=\delta_1 d\delta_2$, where the fixed occurrence of b is in δ_2 and δ_2 contains exactly it b's for an $i \in [1: m-1]$. It follows that there can be at most $2(m-1)\,t$ b's is β, which proves the claim.

Proof of Theorem 4 (continued): Since α is minimal it contains each character u_i and v_i, $i \in [1: n]$ exactly once. Clearly α must have a subsequence γ which is a minimal common supersequence of $\{\alpha_1, \alpha_2, ..., \alpha_n\}$. Since each α_i contains $s_i - 1$ d's, the number of d's in γ is $\leqslant \sum_{i=1}^{n} (s_i - 1) = mt - n$. Let γ' be the subsequence of γ consisting of the characters $b, c_1, c_2, ..., c_r$. Due to the minimality of γ we have $\gamma' = (bc_1c_2...c_r)^p$ for a $p \in [2: mt]$. Assume $p \leqslant mt-1$; then γ has length $\leqslant 2n + (mt-1)(r+1) + mt - n$. Since γ is a minimal common supersequence of $\{\alpha_1, \alpha_2, ..., \alpha_n\}$ and by Claim 1 each minimal common supersequence of $\{\beta_1, \beta_2, ..., \beta_{m-1}\}$ has length $\leqslant 2(m-1)t + m - 1$, we have that α has length $\leqslant 2n + (mt-1)(r+1) + mt - n + 2(m-1)t + m - 1 = t' - r - 1 + 2mt - 2t < t'$ since $r > 2mt - 2t - 1$. Thus we obtain $\gamma' = (bc_1c_2...c_r)\,mt$ and $|\gamma| \leqslant 2n + mt(r+1) + mt - n = n + mt(r+2)$. The minimality of γ now implies that the α_i's can only be embedded into pairwise disjoint substrings of γ. Let γ'' be the subsequence of γ consisting of all b's and d's. Observe that γ'' contains mt b's and that there is a d between at least every second pair of neighboring b's.

Assume that α has an additional b not contained in the subsequence γ''. Let $\hat{\gamma}$ be the sequence obtained after the insertion of this additional b into the corresponding position of γ''. Note that $\hat{\gamma}$ contains exactly $mt+1$ b's. If $\hat{\gamma}$ contains a d between at least every second pair of neighboring b's then it is easily checked that $\hat{\gamma}$ is a supersequence of each sequence β_i, $i \in [1: m-1]$. Otherwise $\hat{\gamma}$ is of the form $\hat{\gamma} = \delta_1\delta_2\delta_3$ with $\delta_2 = dbbbd$ (or $\hat{\gamma} = \delta_1 dbb$ or $\hat{\gamma} = bbd\delta_3$), where δ_1 and δ_3 contain a d between at least every second pair of neighboring b's. Consequently at most one of the strings $\beta_1, \beta_2, ..., \beta_{m-1}$ is not a subsequence of $\hat{\gamma}$. This is the case iff $\hat{\gamma} = \delta_1\, dbbb\, d\delta_3$ and δ_1 contains $it-1$ b's and δ_3 contains $(m-i)\,t-1$ b's for an $i \in [1: m-1]$. Then the string β_i cannot be embedded into $\hat{\gamma}$. It follows that α must have either one additional d between the b's of δ_2 or one additional b to the left or right of δ_2, so that β_i is a subsequence of α. We conclude that if α contains $\geqslant mt+1$ d's then it contains at most two characters more than γ—either two b's or a b and a d—and thus has length $\leqslant |\gamma| + 2 \leqslant n + mt(r+2) + 2 < t'$.

From the above discussion we conclude that α contains each character $b, c_1, c_2, ..., c_r$ exactly mt times. But then α can have length t' only if it contains $mt - n + m - 1$ d's. This is possible only if the subsequence γ'' of α which contains $mt - n$ d's has no d between the (it)th and $(it+1)$th b for $i \in [1: m-1]$. Then α must have an additional d between the (it)th and $(it+1)$th b for $i \in [1: m-1]$ because otherwise the string β_i is not a subsequence of α. Let $\alpha'_i = (bd)^{s}_{i-1}\, b$ for $i \in [1: n]$. Now we have $\gamma'' = \alpha'_{\pi(1)}\alpha'_{\pi(2)}...\alpha'_{\pi(n)}$ for a permutation π of $[1: n]$ such that $\alpha'_{\pi(3i-2)}\alpha'_{\pi(3i-1)}\alpha'_{\pi(3i)}$ has n b's for $i \in [1: m]$. It follows that $\{A_1, A_2, ..., A_m\}$ with $A_i = \{s_{\pi(3i-2)}, s_{\pi(3i-1)}, s_{\pi(3i)}\}$ for $i \in [1: m]$ is the partition of A which was sought. ▌

The LMCS Problem for Strings of Length 2. We now describe a linear-time algorithm to determine an lmcs for strings of length 2. Let S be a set of strings each of length 2. Let $G = (V, E)$ be the corresponding directed graph where V is the alphabet and $(a, b) \in E$ iff $ab \in S$. For ease of description we first assume that each string in S is of the form ab with $a \neq b$, which means G has no loops. We further assume that G has no isolated nodes. The algorithm is as follows:

(1) Compute the strongly connected components of G (recall that a directed graph is strongly connected if for every two different nodes u, v there exists a directed path from u to v as well as from v to u). Represent each strongly connected component by any of its nodes and let $V^* \subset V$ be the set of all representatives of the strongly connected components of G. Let $G^* = (V^*, E^*)$ be the directed graph with $(a, b) \in E^*$ iff there exist nodes c and d in the strongly connected components represented by a and b respectively such that $(c, d) \in E$. Clearly G^* contains no directed cycle. Let $V_{\text{sou}} = \{v_1, v_2, ..., v_p\} \subset V^*$ and $V_{\text{sin}} = \{v_{p+1}, v_{p+2}, ..., v_q\} \subset V^*$ be the sets of sources and sinks respectively in G^* (note that G^* may have isolated nodes which we include in V_{sin} but not in V_{sou}). Set $V' = V_{\text{sou}} \cup V_{\text{sin}}$.

(2) Set $W = V'$ and $\alpha = v_1 v_2 ... v_q$.

(3) WHILE $V \neq W$ DO.

Compute a directed path or cycle $w_0, w_1, ..., w_r$ in G with $w_0, w_r \in W$ and $w_1 w_2 ... w_{r-1} \in V - W$. Set

$$\alpha = w_{r-1} w_{r-2} ... w_1 \alpha w_{r-1} w_{r-2} ... w_1$$

and

$$W = W \cup \{w_1, w_2, ..., w_{r-1}\}.$$

(4) Return α.

Clearly steps (2) and (4) can be done in linear time. To see that step (1) can be done in linear time recall that the strongly connected components of a directed graph can be found in linear time. Also observe that if in step (3) $V \neq W$ it is always possible to find in linear time the required directed path or cycle containing at least one node in $V - W$. Clearly the loop in step (3) is executed at most $|V|$ times. It is not hard to find a linear time implementation of step (3) but we omit the details here. It follows that the entire algorithm runs in linear time. It remains to establish correctness.

The string α returned by the algorithm has the form $\beta\gamma\beta$, where $\gamma = v_1 v_2 \ldots v_q$ contains each character in V' and where β is a permutation of the characters in $V - V'$. Hence α contains each string of the form ab with $a \in V - V'$ or $b \in V - V'$. For each string $ab \in S$ with $a, b \in V'$ we have that $a \in V_{\text{sou}}$ and $b \in V_{\text{sin}}$. Since each string ab with $a \in V_{\text{sou}}$ and $b \in V_{\text{sin}}$ is a subsequence of γ we conclude that α is a common supersequence of S.

Now we show that α is minimal. Since G has no isolated nodes each character in V must be contained in α. Since the characters in V' are contained only once in α none of their occurrences can be omitted. All characters in $V - V'$ are contained exactly two times in α. Consider step (3) of the algorithm when the directed path or cycle $w_0, w_1, \ldots, w_r$ is identified and the new string $\alpha = w_{r-1} w_{r-2} \ldots w_1 \alpha w_{r-1} w_{r-2} \ldots w_1$ is formed. Due to the edge $(w_{r-1}, w_r) \in E$, and since w_r is contained only in the old string α, the left occurrence of w_{r-1} cannot be omitted. For each $i \in [1: k-2]$ due to the edge $(w_i, w_{i+1}) \in E$ the left occurrence of w_i and the right occurrence of w_{i+1} cannot be omitted. Finally due to the edge $(w_0, w_1) \in E$ the right occurrence of w_1 cannot, be omitted. So we have shown that α is a minimal supersequence of S. Before we can establish that α is an lmcs of S we need the following lemma.

Lemma 1 *Let S be a set of strings each of length 2 where each string in S is of the form ab with $a \neq b$. Then for every common supersequence α of S which contains every character at least twice it is possible to omit the leftmost occurrence of one of the characters in α such that the sequence so obtained is a common supersequence of S. By symmetry, the same holds with "rightmost" instead of "leftmost."*

Proof. Assume for a contradiction that there exists a common supersequence α of S which contains every character at least twice and such that, for each character, if its left occurrence is omitted, the string so obtained is not a common supersequence. Without loss of generality assume further that each character occurs not more than twice in α (otherwise all occurrences between the leftmost and the rightmost occurrence can be omitted). Let a be the leftmost character in S. Then there must exist a character b which has both occurrences between the a's in α, i.e.,

$$\alpha = a \ldots b \ldots b \ldots a \ldots$$

This holds because otherwise the left occurrence of a can be omitted. Let b be such that (i) no character occurs twice between the occurrences of b and (ii) no character has both its occurrences to the left of the leftmost occurrence of b. Now there must be a character c which has one occurrence to the left of the two b's and the other between the two b's in α, i.e.,

$$\alpha = a \ldots c \ldots b \ldots c \ldots b \ldots a \ldots$$

This holds because otherwise the leftmost occurrence of b can be omitted. Furthermore let c be such that no other character has both its occurrences between the occurrences of c. Iterating this argument shows the existence of a character d which has one occurrence between the occurrences of c and the other one to the left of the leftmost occurrence of c and so forth. But this is not possible since α is finite. ▮

Now we establish that α is an lmcs for S. For a contradiction assume that there exists a minimal common supersequence β of S which is longer than α. Consequently there must exist a source or sink in G^* such that all characters included in this component occur twice in β. Without loss of generality let this component be a sink. Let $U \subset V$ be the set of characters contained in this component and let $T \subset S$ be the set of strings in S having both characters in U. Let γ be the subsequence of β containing exactly the characters in U. Lemma 1 implies that we can omit one of the leftmost occurrences of a character in γ and still have a common supersequence of T. Let β' be the string that is obtained if we omit the corresponding occurrence of a character in β. Since the characters in U are contained in a sink all strings in $S - T$ which contain a character in U are of the form ab with $a \in V - U$ and $b \in U$. Thus β' must be a common supersequence of S which contradicts the minimality of β.

Recall that it was assumed that S contains no strings of the form aa and thus G has no loops. If this restriction is omitted the algorithm is changed as follows. If in step (1) a strongly connected component contains characters with a loop then the component is represented by one of these characters. In step (2) we add a second occurrence for those characters in V' with a loop. Simple modifications of the proof given above show the correctness of this algorithm. Thus the following theorem is proved.

Theorem 5. *The LMCS problem for strings of length 2 is solvable in linear time.*

6. CONCLUSION AND OPEN PROBLEMS

We have shown that, in the case of two strings (or indeed any fixed number of strings), a shortest maximal common subsequence and a longest minimal common supersequence can be found in polynomial time by dynamic programming. However, for general k, we have shown that finding a shortest maximal common subsequence or a longest minimal common supersequence of k strings is NP-hard. Further, unless P = NP, the length of a shortest maximal common subsequence cannot be approximated in polynomial time, within a factor of k^{δ} for any $\delta < 1$.

It is natural to conjecture that finding a good approximation to the length of a longest minimal common supersequence of k strings is just as hard, but we have no result of this kind.

Finally, the problem of finding a longest minimal common supersequence in the case of strings of length 2 is shown to be solvable in polynomial time, in contrast to finding the shortest common supersequence, which is NP-hard in this case. The LMCS problem in the case of strings of fixed length > 2 remains open.

Received September 30, 1994; in revised form August 31, 1995.

REFERENCES

[AG87] Apostolico, A. and Guerra, C. (1987), The longest common subsequence problem revisited, *Algorithmica* **2**, 315–336.

[GJ79] Garey, M. R. and Johnson, D. S. (1979),"Computers and Intractability," Freeman, San Francisco.

[Hal93] Halldórsson, M. M. (1993), Approximating the minimum maximal independence number, *Inform. Process. Lett.* **46**, 169–172.

[Hir75] Hirschberg, D. S. (1975), A linear space algorithm for computing maximal common subsequences, *Comm. ACM* **18**, 341–343.

[Hir77] Hirschberg, D. S. (1977), Algorithms for the longest common subsequence problem, *J. Assoc. Comput. Mach.* **24**, 664–675.

[HS77] Hunt, J. W. and Szymanski, T. G. (1977), A fast algorithm for computing longest common subsequences, *Comm. ACM* **20**, 350–353.

[Irv91] Irving, R. W. (1991), On approximating the minimum independent dominating set, *Inform. Process. Lett.* **37**, 197–200.

[JL94] Jiang, T. and Li, M. (1994), On the approximation of shortest common supersequences and longest common subsequences, *in* "Proceedings of the 21st International Colloquium on Automata, Languages and Programming, Jerusalem 1994." Springer-Verlag LNCS, Vol. 820, pp. 191–202.

[Mai78] Maier, D. (1978), The complexity of some problems on subsequences and supersequences, *J. Assoc. Comput. Mach.* **25**, 322–336.

[Mid92] Middendorf, M. (1992), "Zur Komplexität von Einbettungsproblemen für Wortmengen," Ph.D. Thesis, Fachbereich Mathematik, Universität Hannover.

[PY91] Papadimitriou, C. H. and Yannakakis, M. (1991), Optimization, approximation, and complexity classes, *J. Comput. System Sci.* **43**, 425–440.

[RU81] Räihä, K.-J., and Ukkonen, E. (1981), The shortest common supersequence problem over binary alphabet is NP-complete, *Theoret. Comp. Sci.* **16**, 187–198.

[Tim89] Timkovskii, V. G. (1989), Complexity of common subsequence and supersequence problems and related problems, English translation from *Kibernetika* **5**, 1–13.

[Ukk85] Ukkonen, E. (1985), Algorithms for approximate string matching, *Inform. and Control* **64**, 100–118.

[WMMM90] Wu, S., Manber, U., Myers, G., and Miller, W. (1990), An $O(NP)$ sequence comparison algorithm, *Inform. Process. Lett.* **35**, 317–323.

[YG80] Yannakakis, M. and Gavril, F. (1980), Edge dominating sets in graphs, *SIAM J. Appl. Math.* **38**, 364–372.