Polynomial-Delay Enumeration of Maximal Common Subsequences

Alessio Conte[1], Roberto Grossi[1], Giulia Punzi[1](✉), and Takeaki Uno[2]

[1] Università di Pisa, Pisa, Italy
{conte,grossi}@di.unipi.it, giuliagpunzi@gmail.com
[2] National Institute of Informatics, Tokyo, Japan
uno@nii.ac.jp

Abstract. A *Maximal Common Subsequence* (MCS) between two strings X and Y is an inclusion-maximal subsequence of both X and Y. MCSs are a natural generalization of the classical concept of Longest Common Subsequence (LCS), which can be seen as a longest MCS. We study the problem of efficiently listing all the *distinct* MCSs between two strings. As discussed in the paper, this problem is algorithmically challenging as the same MCS cannot be listed multiple times: for example, dynamic programming [Fraser et al., CPM 1998] incurs in an exponential waste of time, and a recent algorithm for finding an MCS [Sakai, CPM 2018] does not seem to immediately extend to listing. We follow an alternative and novel graph-based approach, proposing the first output-sensitive algorithm for this problem: it takes polynomial time in n per MCS found, where $n = \max\{|X|, |Y|\}$, with polynomial preprocessing time and space.

1 Introduction

The widely known Longest Common Subsequence (LCS) is a special case of the general notion of (inclusion-)Maximal Common Subsequence (MCS) between two strings X and Y. Defined formally below, the MCS is a subsequence S of both X and Y such that inserting any character at any position of S no longer yields a common subsequence. We believe that the enumeration of the distinct MCSs is an intriguing problem from the point of view of string algorithms, for which we offer a novel graph-theoretic approach in this paper.

Problem Definition. Let Σ be an alphabet of size σ. A string S over Σ is a concatenation of any number of its characters. A string S is a *subsequence* of a string X, denoted $S \subset X$, if there exist $0 \leq i_0 < \dots < i_{|S|-1} < |X|$ such that $X[i_k] = S[k]$ for all $k \in [0, |S| - 1]$.

Definition 1. *Given two strings X, Y, a string S is a* Maximal Common Subsequence *of X and Y, denoted $S \in MCS(X, Y)$, if*

1. $S \subset X$ and $S \subset Y$; that is, S is a common subsequence;

2. *there is no other string W satisfying the above condition 1 such that $S \subset W$, namely, S is inclusion-maximal as a common subsequence.*

Example 1. Consider $X = \texttt{TGACGA}$ and $Y = \texttt{ATCGTA}$, where $MCS(X,Y) = \{\texttt{TCGA}, \texttt{ACGA}\}$. A greedy left-to-right common sequence is not necessarily a MCS: iteratively finding the nearest common character in X and Y, from left to right, gives $W = \texttt{TGA}$, which is not in $MCS(X,Y)$ as $\texttt{TGA} \subset \texttt{TCGA}$.

The focus of this paper is on the enumeration of $MCS(X,Y)$ between two strings X and Y, stated formally below.

*Problem 1 (***MCS enumeration***).* Given two strings X, Y of length $O(n)$ over an alphabet Σ of size σ, list all maximal common subsequences $S \in MCS(X,Y)$.

In enumeration algorithms, the aim is to list all objects of a given set. The time complexity of these type of algorithms depends on the cardinality of the set, which is often exponential in the size of the input. This motivates the need to define a different complexity class, based on the time required to output one solution.

Definition 2. *An enumeration algorithm is* polynomial delay *if it generates the solutions one after the other, in such a way that the delay between the output of any two consecutive solutions is bounded by a polynomial in the input size.*

Our aim will be to provide a polynomial delay MCS enumeration algorithm, more specifically we will prove the following result.

Theorem 1. *There is a polynomial-delay enumeration algorithm for Problem 1, with polynomial preprocessing time and space.*

In the full version of this paper, we show how to get a small $O(n\sigma(\sigma + \log n))$ polynomial delay, with $O(n^2(\sigma + \log n))$ preprocessing time and $O(n^2)$ space.

Motivation and Relation to Previous Work. Maximal common subsequences were first introduced in the mid 90s by Fraser et al. [5]. Here, the concept of MCS was a stepping stone for one of the main problems addressed by the authors: the computation of the length of the Shortest Maximal Common Subsequence (SMCS) (i.e. the shortest string length in $MCS(X,Y)$), introduced in the context of LCS approximation. For this, a dynamic programming algorithm was given to find the length of a SMCS of two strings in cubic time.

While LCSs have thoroughly been studied [3,6,10,12], little is known for MCSs. In general, LCSs only provide with information about the longest possible alignment between two strings, while MCSs offer a different range of information, possibly revealing slightly smaller but alternative alignments. A recent paper by Sakai [11] presents an algorithm that deterministically finds one MCS between two strings of length $O(n)$ in $O(n \log\log n)$ time, in contrast with the computation of the length of the LCS, for which a quadratic conditional lower bound based on SETH has been proved [1]. This same algorithm can also be

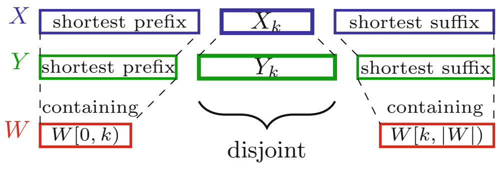


Fig. 1. Visual representation of Sakai's characterization

used to extend a given sequence to a maximal one in the same time. Furthermore, an $O(n)$-time algorithm to check whether a given subsequence is maximal is described in the paper. To this end, Sakai gives a neat characterization of MCSs, which will be useful later, as stated in Lemma 1 and illustrated in Fig. 1.

Lemma 1 (MCS Characterization [11]**).** *Given a common subsequence W of X and Y, we define X_k (resp. Y_k) as the remaining substring obtained from X (resp. Y) by deleting both the shortest prefix containing $W[0,k)$, and the shortest suffix containing $W[k,|W|)$. Substrings X_k, Y_k are called the k-th* interspaces. *With this notion, W is maximal if and only if $X_k \cap Y_k = \emptyset$ for all $k \in [0,|W|]$.*

The aforementioned two results seem to be of little help in our case, as neither of the two can be directly employed to obtain a polynomial-delay enumeration algorithm to solve Problem 1, which poses a quite natural question.

Consider the dynamic programming approach in [5]: even if the dynamic programming table can be modified to list the lengths of all MCS in polynomial time, this result cannot be easily generalized to Problem 1. Indeed we show below that any incremental approach, including dynamic programming, leads to an *exponential*-delay enumeration algorithm.

Example 2. Consider $X = \texttt{TAATAATAAT}$, $Z = \texttt{TATATATATAT}$ and $Y = Z\,Z$. Since $X \subset Y$, the only string in $MCS(X,Y)$ is the whole X. But if we were to proceed incrementally over Y, at half way we would compute $MCS(X,Z)$, which can be shown to have size $O(\exp(|X|))$. This means that it would require an exponential time in the size of the input to provide just a single solution as output.

As for the approach in [11], the algorithm cannot be easily generalized to solve Problem 1, since the specific choices it makes are crucial to ensure maximality of the output, and the direct iterated application of Lemma 1 does not lead to an efficient algorithm for Problem 1, as shown next.

Example 3. For a given common subsequence W to start with, first find all values of $k \leq |W|$ such that $X_k \cap Y_k \neq \emptyset$. Then, for these values, compute all distinct characters $c \in X_k \cap Y_k$, and for each of these recur on the extended sequences $W' = W[0,...,k-1]\,c\,W[k,...,|W|-1]$. For instance, given the strings $X = \texttt{ACACA}$ and $Y = \texttt{ACACACA}$ with starting sequences $W = \texttt{A}$ and $W = \texttt{C}$, this algorithm would recur on almost every subsequence of X, just to end up outputting the single $MCS(X,Y) = \{X\}$ an exponential (in the size of X) number of times.

Getting polynomial-delay enumeration is therefore an intriguing question. The fact that one maximal solution can be found in polynomial time does not directly imply an enumeration algorithm: there are cases where this is possible, but the existence of an output-sensitive enumeration algorithm is an open problem [8], or would imply $P = NP$ [9]. As we will see, solving Problem 1 can lead to further pitfalls that we circumvent in this paper.

2 MCS as a Graph Problem

As a starting point we reduce Problem 1 to a graph problem in order to give a theoretic characterization and get some insight on how to combine MCS. Afterwards, this characterization will be reformulated in an operative way, leading to an algorithm for MCS enumeration.

2.1 Graph $G(X, Y)$

Definition 3. *Given two strings X, Y, their* string bipartite graph *$G(X, Y)$ has vertex set $V = [0, |X| - 1] \cup [0, |Y| - 1]$ representing the positions inside X and Y, and edge set $E = \{(i, j) \mid X[i] = Y[j]\}$ where each edge, called* pairwise occurrence*, connects positions with the same character in different strings.*

Definition 4. *A* mapping *of $G(X, Y)$ is a subset $\mathcal{P}$ of its edges such that for any two edges $(i, j), (h, k) \in \mathcal{P}$ we have $i < h$ iff $j < k$. That is, a mapping is a non-crossing matching of the string graph.*

Each mapping of the string graph spells a common subsequence. Vice versa, each common subsequence has at least one corresponding mapping. Thus one might incorrectly think that MCS correspond to inclusion-maximal mappings; as a counterexample consider $X = \mathtt{AGG}$ and $Y = \mathtt{AGAG}$, with $MCS(X, Y) = \{\mathtt{AGG}\}$. $G(X, Y)$ has an inclusion-maximal mapping corresponding to $\mathtt{AG} \notin MCS(X, Y)$.

For a string S, let $next_S(i)$ be the smallest $j > i$ with $S[j] = S[i]$ (if any), and $next_S(i) = |S| - 1$ otherwise; we use the shorthand $\mathcal{I}_S(i) = S[i+1, \ldots, next_S(i)]$.

Definition 5. *A mapping of $G(X, Y)$ is called* rightmost *if for each edge (i, j) of the mapping, corresponding to character $c \in \Sigma$, the next edge (i', j') of the mapping is such that $next_X(i) \geq i'$ and $next_Y(j) \geq j'$. That is, there are no occurrences of c in $X[i+1, \ldots, i'-1]$ and $Y[j+1, \ldots, j'-1]$, the portions between edges (i, j) and (i', j'). We can symmetrically define a* leftmost *mapping.*

In order to design an efficient and correct enumeration algorithm that uses also Definition 5, we first need to study how $MCS(X, Y)$ and $MCS(X', Y')$ relate to $MCS(X X', Y Y')$ for any two pairs of strings X, Y and X', Y'.

Remark 1. A simple concatenation of the pairwise MCS fails: consider for example $X = \mathtt{AGA}$, $X' = \mathtt{TGA}$, $Y = \mathtt{TAG}$ and $Y' = \mathtt{GAT}$, with $MCS(X X', Y Y') = \{\mathtt{AGGA}, \mathtt{AGAT}, \mathtt{TGA}\}$. We have $MCS(X, Y) = \{\mathtt{AG}\}$ and $MCS(X', Y') = \{\mathtt{GA}, \mathtt{T}\}$. Combining the latter two sets we find the sequence $\mathtt{AGGA}$, which is in fact maximal, but also $\mathtt{AGT}$, which is not maximal as $\mathtt{AGT} \subset \mathtt{AGAT}$.

The correct condition for combining MCS is a bit more sophisticated, as stated in Theorem 2. Here, for a position i of a string S, we denote by $S_{<i}$ the prefix of S up to position $i-1$, and by $S_{>i}$ the suffix of S from position $i+1$.

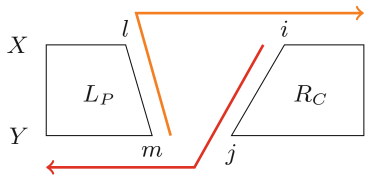


Fig. 2. For their concatenation to be a MCS, P has to be maximal in the red part and C in the orange one (Color figure online)

Theorem 2. *(MCS Combination) Let P and C be common subsequences of X, Y. Let (l, m) be the last edge of the* leftmost *mapping L_P of P, and (i, j) be the first edge of the* rightmost *mapping R_C of C (see Fig. 2). Then*

$$PC \in MCS(X, Y) \text{ iff } P \in MCS(X_{<i}, Y_{<j}) \text{ and } C \in MCS(X_{>l}, Y_{>m}).$$

Proof. To ensure the equivalence, it is sufficient to show that Sakai's interspaces for string PC over X, Y are the same as the ones for either P over $X_{<i}, Y_{<j}$, or for C over $X_{>l}, Y_{>m}$. Let $|P| = s$, $|C| = r$, and consider an index k.

Case $k < s$: the shortest suffixes of X, Y containing $p_{k+1}, ..., p_s, c_1, ..., c_r$ are unchanged from the shortest suffixes of $X_{<i}, Y_{<j}$ containing $p_{k+1}, ..., p_s$, since C is already in rightmost form starting exactly at (i, j). The shortest prefixes containing $p_1, ..., p_k$ are simply the first k edges of the leftmost mapping of P, both in X, Y and $X_{<i}, Y_{<j}$. Therefore, the interspaces for the whole strings are unchanged from the interspaces for P over $X_{<i}, Y_{<j}$.

Case $k > s$: this case is symmetrical to the previous one: the interspaces for the whole strings are unchanged from the interspaces for C over $X_{>l}, Y_{>m}$.

Case $k = s$: The last interspaces for P and the first for C coincide, and they are $X[l+1, ..., i-1]$ and $Y[m+1, ..., j-1]$. Since P is in leftmost form ending at (l, m) and C is in rightmost form beginning at (i, j), these two strings also coincide with the k-th interspaces for PC. $\square$

Theorem 2 gives a precise characterization on how to combine maximal subsequences, but it cannot be blindly employed to design an enumeration algorithm for a number of reasons.

Let a string P be called a *valid prefix* if there exists $W \in MCS(X, Y)$ such that P is a prefix of W. Suppose that the leftmost mapping for P ends with edge (l, m), and that we want to expand P by appending characters to it so that it remains valid. These characters correspond to the edges (i, j) related to (l, m) as stated by Theorem 2, for some maximal sequence C. The rest of the paper describes how to perform this task without explicitly knowing C.

Remark 2. It does not work to consider every edge (i, j) satisfying the first condition in Theorem 2, that is, $P \in MCS(X_{<i}, Y_{<j})$. As a counterexample,

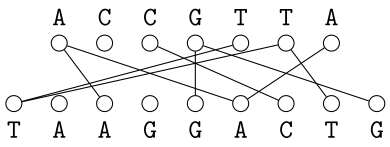
consider $X = \texttt{AGAGAT}$ and $Y = \texttt{TAGGA}$. Note that $P = \texttt{AG}$ is a valid prefix, since $\texttt{AGGA} \in MCS(X, Y)$, and its leftmost mapping ends at edge $(l, m) = (1, 2)$, labeled with $\texttt{G}$. Edge $(i, j) = (2, 4)$ corresponding to character $\texttt{A}$ is such that $P \in MCS(X_{<i}, Y_{<j})$; but $P\,\texttt{A} = \texttt{AGA}$ is not a valid prefix (and $\texttt{AGA} \subset \texttt{AGGA}$). Along the same lines, Sakai's algorithm cannot help here. It generates a MCS that contains P as a subsequence, but *not* necessarily as a prefix. Therefore, it cannot be easily employed to identify the edges (i, j).

We need a more in-depth study of the properties of graph $G(X, Y)$ to characterize the relationship between (l, m) and (i, j). First, we give the notion of *unshiftable edges*, and show that edge (i, j) needs to be unshiftable. Second, as being unshifable is only a necessary condition, we discuss how to single out the (i, j)'s suitable for our given (l, m).

2.2 Unshiftable Edges

Definition 6. *An edge (i, j) of the bipartite graph $G(X, Y)$ is called* unshiftable *if it belongs to at least one maximal rightmost mapping of $G(X, Y)$. The set of unshiftable edges is denoted $\mathcal{U}$. An edge is called* shiftable *if it is not unshiftable.*

Example 4. Consider $X = \texttt{ACCGTTA}$ and $Y = \texttt{TAAGGACTG}$. The unshiftable edges for these two strings are the following ones.



Unshiftable edges[1] can be characterized in a more immediate way, stated in Proposition 1.

Proposition 1. *An edge (i, j) is unshiftable if and only if either (i) it corresponds to the rightmost pairwise occurrence of $X[i] = Y[j]$ in the strings, or (ii) there is at least one unshiftable edge in the subgraph $G(\mathcal{I}_X(i), \mathcal{I}_Y(j))$.*

Proof. (*Only if*) It is sufficient to show that all edges of a maximal rightmost mapping satisfy one of the two conditions. Let $\mathcal{R} = r_1, ..., r_N$ be a rightmost maximal mapping of $G(X, Y)$. By definition of rightmost mapping, r_N corresponds to the last pairwise occurrence of some character, thus it satisfies the base case. Consider now $p < N$, and let r_p correspond to some character c. By definition of unshiftability, $r_k \in \mathcal{U}$ for all k, specifically $r_{p+1} \in \mathcal{U}$. Since the mapping is rightmost maximal, there can be no occurrences of c between r_p and r_{p+1}; therefore the unshiftable edge r_{p+1} belongs to the subgraph $G(\mathcal{I}_X(i), \mathcal{I}_Y(j))$.

[1] A symmetric definition of *left*-unshiftable edges can be given by considering maximal leftmost mappings. The k-dominant edges for LCS [2,4,7] are a subset of left-unshiftable edges.

(*If*) Let (i, j) satisfy one of the two conditions. If it satisfies the base case, then (i, j) is in rightmost form, and we can extend it to the left to a rightmost maximal mapping. On the other hand, let (i, j) satisfy the second condition. Then, there is an edge $(h, k) \in \mathcal{U}$ that belongs to the subgraph $G(\mathcal{I}_X(i), \mathcal{I}_Y(j))$. Consider the rightmost maximal mapping $\mathcal{R}$ that contains (h, k); if it also contains (i, j) we are done. Otherwise, let $\mathcal{R}' \subset \mathcal{R}$ be the restriction that only contains (h, k) and subsequent edges. Consider the rightmost mapping $(i, j) \cup \mathcal{R}'$; we can extend it to the left until it is rightmost maximal. In any case, we have obtained a rightmost maximal mapping containing (i, j), which is then unshiftable. □

Remark 3. Although every MCS has a corresponding rightmost maximal mapping, and the edges in the latter are unshiftable by Definition 6, it is incorrect to conclude that the opposite holds too. Not all rightmost maximal mappings give MCS: consider for example $X =$ `AAGAAG` and $Y =$ `AAGA`. In $G(X, Y)$ we have a maximal rightmost mapping for `AAG`, but `AAG` $\subset$ `AAGA` $\in MCS(X, Y)$.

2.3 Candidate Extensions

We finalize the characterization of the relationship between edges (l, m) and (i, j) of Theorem 2, where (l, m) is the last edge of the leftmost mapping in $G(X, Y)$ of a valid prefix P. We would like to single out a priori the corresponding possible (i, j)'s, without explicitly knowing their Cs. This in turn will lead to the incremental discovery of such Cs one character c at a time.

Specifically, we look for edges (i, j) corresponding to the characters $c \in \Sigma$ such that $P\,c$ is still a valid prefix.

Definition 7. *Given an edge (l, m), its* cross $\chi_{(l,m)} = \langle e, f \rangle$ *(see Fig. 3) is given by (at most) two unshiftable edges $e = (e_1, e_2)$, $f = (f_1, f_2)$ such that*

$$e_1 = \min\{h_1 > l \mid \exists h_2 > m : (h_1, h_2) \in \mathcal{U}\} \text{ and } e_2 = \min\{h_2 > m : (e_1, h_2) \in \mathcal{U}\},$$

$$f_2 = \min\{h_2 > m \mid \exists h_1 > l : (h_1, h_2) \in \mathcal{U}\} \text{ and } f_1 = \min\{h_1 > l : (h_1, f_2) \in \mathcal{U}\}.$$

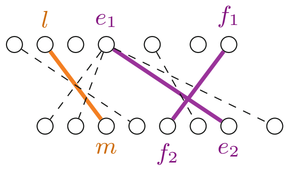


Fig. 3. Graphical representation of the cross $\langle e, f \rangle$ for edge (l, m), drawn in purple: $e = (e_1, e_2)$, $f = (f_1, f_2)$ are the first unshiftable edges soon after (l, m). (Color figure online)

Definition 8. *Given an edge (l, m), let $\chi_{(l,m)} = \langle e, f\rangle$ be its cross. We define the set of its* mikado edges *as the unshiftable edges of $G(X[e_1, ..., f_1], Y[f_2, ..., e_2])$,*

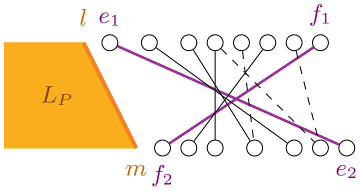
$$Mk_{(l,m)} = \{(i, j) \in \mathcal{U} \mid e_1 \leq i \leq f_1 \text{ and } f_2 \leq j \leq e_2\},$$

and the subset of candidate extensions *for (l, m) as*

$$Ext_{(l,m)} = \{(i, j) \in Mk_{(l,m)} \mid \nexists (h, k) \in Mk_{(l,m)} \setminus (i, j) \text{ such that } h \leq i \text{ and } k \leq j\}.$$

It follows immediately from the definition that no two edges in $Ext_{(l,m)}$ have a common endpoint, and thus $|Ext_{(l,m)}| \leq n$.

Definitions 7 and 8 find their application in identifying a valid prefix extension, as shown in Fig. 4 and discussed next.



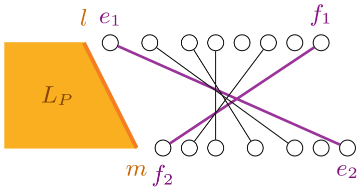
$\longrightarrow$


Fig. 4. Extraction of $Ext_{(l,m)}$ from the set $Mk_{(l,m)}$, pictured on the left. The edges belonging to $Mk_{(l,m)} \setminus Ext_{(l,m)}$ are dashed.

2.4 Valid Prefix Extensions

Let P be a valid prefix with leftmost mapping L_P ending at edge (l, m). We use shorthands for $Mk_P = Mk_{(l,m)}$ and $Ext_P = Ext_{(l,m)}$. The candidates in $Ext_P \subseteq Mk_P$ are the unshiftable edges soon after L_P such that no other unshiftable edge lies completely delimited between L_P and any of them, as illustrated in Fig. 4.

We thus are ready to give our algorithmic characterization of valid extensions of prefixes to relate edges (l, m) and (i, j) from Theorem 2.

Theorem 3. *Let P be a valid prefix of some $M \in MCS(X, Y)$, with leftmost mapping L_P ending at edge (l, m). Then $P\,c$ is a valid prefix if and only if the following two conditions hold.*

(1) There exists $(i, j) \in Ext_P$ corresponding to character c, and
(2) $P \in MCS(X_{<i}, Y_{<j})$.

The proof of Theorem 3 is quite involved, and thus postponed to Sect. 4. This result is crucial for our polynomial-delay binary partition algorithm, as the latter recursively enumerates $MCS(X, Y)$ by building increasingly long valid prefixes and avoiding unfruitful recursive calls.

3 Polynomial-Delay MCS Enumeration Algorithm

The characterization given in Theorem 3 immediately gives prefix-expanding enumeration Algorithm 1, which progressively augments prefixes with characters that keep them valid, until whole MCSs are recursively generated. It is worth noting that Theorem 3 guarantees that each recursive call yields at least one MCS; moreover, all the MCSs are listed once.

Algorithm 1 employs a binary partition scheme. First, it builds the necessary data structures and finds the set of unshiftable edges in a polynomial preprocessing phase, using FindUnshiftables as described in detail in Sect. 3.1. Then, it begins a recursive computation BinaryPartition where, at each step, it considers the enumeration of the MCSs that start with some valid prefix P. The partition is made through over characters $c \in \Sigma$ such that $P\,c$ is valid.

For convenience, we add a dummy character $\# \notin \Sigma$ at the beginning of both strings; i.e. at positions $(-1,-1)$. The recursive computation then starts with $P = \#$, and leftmost mapping $L_P = \{(-1,-1)\}$. At each step, the procedure

Algorithm 1. EnumerateMCS

```
 1: procedure EnumerateMCS($X$, $Y$, $\Sigma$)
 2:     $\mathcal{U}$ = FindUnshiftables(($|X|$, $|Y|$))
 3:     BinaryPartition($\#$, $\{(-1,-1)\}$)
 4: end procedure

 5: procedure FindUnshiftables($(i,j)$)
 6:     for $c \in \Sigma$ do
 7:         $l_X \leftarrow$ the right-most occurrence of $c$ in $X_{<i}$
 8:         $l_Y \leftarrow$ the right-most occurrence of $c$ in $Y_{<j}$
 9:         if $l_X \neq -1$ and $l_Y \neq -1$ and $(l_X, l_Y) \notin \mathcal{U}$ then
10:             yield $(l_X, l_Y)$        // which is added to $\mathcal{U}$
11:             FindUnshiftables($(l_X, l_Y)$)
12:         end if
13:     end for
14: end procedure

15: procedure BinaryPartition($P$, $L_P$)
16:     compute the set of extensions $Ext_P$ using $\mathcal{U}$
17:     if $Ext_P = \emptyset$ then Output $P$
18:     else
19:         for $(i,j) \in Ext_P$ corresponding to some $c \in \Sigma$ do
20:             if $P \in MCS(X_{<i}, Y_{<j})$ then
21:                 let $(l,m)$ be the last edge of $L_P$
22:                 find leftmost mapping edge $(l_c, m_c)$ for $c$ in $G(X_{>l}, Y_{>m})$
23:                 BinaryPartition($P\,c$, $L_P \cup (l_c, m_c)$)
24:             end if
25:         end for
26:     end if
27: end procedure
```

finds the valid extensions Ext_P for the given prefix P using the unshiftable edges from $\mathcal{U}$. If Ext_P is empty, then P is an MCS, and is returned. Otherwise, for each character $c \in \Sigma$ corresponding to an edge in Ext_P (i.e. condition *(1)* of Theorem 3), it checks whether $P\,c$ satisfies condition *(2)* of Theorem 3. If it does, given the last edge (l, m) of L_P, the algorithm finds the leftmost mapping (l_c, m_c) for character c in $G(X_{>l}, Y_{>m})$, as to update $L_{P\,c} = L_P \cup (l_c, m_c)$. Then, it partitions the MCSs to enumerate into the ones that have $P\,c$ as a prefix, and recursively proceeds on $P\,c$, and $L_P \cup (l_c, m_c)$. The correctness of Algorithm 1 immediately follows from Theorem 3.

3.1 Finding Unshiftable Edges

We compute unshiftable edges by going backwards in the strings X and Y, and exploiting the observation below, which follows immediately from Proposition 1.

Fact 1. *Let $(i, j) \in \mathcal{U}$ and $c \in \Sigma$. If i', j' are the rightmost occurrences of c respectively in $X_{<i}$ and $Y_{<j}$, then edge (i', j') is also unshiftable.*

Symmetrically as we did in the previous section, let us add a special character $\$ \notin \Sigma$ at the end of both strings, as to obtain an unshiftable edge at the last positions $(|X|, |Y|)$. Starting from this edge, we have a natural recursive visiting procedure that finds unshiftable edges based on the Fact 1. For each character $c \in \Sigma$, candidate unshiftable edges are found by taking the rightmost occurrences of c before the current edge in both strings. Then, we recur in these new edges, unless already visited. This originates our FindUnshiftables procedure, whose pseudocode is shown in Algorithm 1.

All unshiftable edges are found in this fashion. The last pairwise occurrences of every character are visited from edge $(|X|, |Y|)$. If an unshiftable edge (i, j) is not the last pairwise occurrence, then by Proposition 1 there is at least one unshiftable edge in $G(\mathcal{I}_X(i), \mathcal{I}_Y(j))$. Edge (i, j) will then surely be visited from the leftmost of these edges, and therefore it will be marked as unshiftable.

3.2 Polynomial Delay

We now show that Algorithm 1 has polynomial delay by analyzing the two main components of EnumerateMCS.

The following remark is crucial for our complexity analysis:

Remark 4. Unshiftable edges can be dense in $G(X, Y)$. For example, consider $X = \mathtt{A}^n(\mathtt{CA})^n$ and $Y = \mathtt{A}^n\mathtt{C}^n$. In this situation, every $\mathtt{A}$ of Y has out-degree of unshiftable edges equal to the number of $\mathtt{C}$s in X, that is $O(n)$. The total number of unshiftable edges is therefore $|\mathcal{U}| = O(n \cdot n) = O(n^2)$.

In the rest of the section, we assume that the next and previous occurrences of a given character with respect to some position of the strings can be performed in logarithmic time, with a linear-space search tree.

Preprocessing Phase of FindUnshiftables. This algorithm examines every unshiftable edge exactly once, adding it to a set if not already found (see Line 9). For each of these edges it finds the previous pairwise occurrences of every character and checks whether they are already in the unshiftable set. This takes $O(|\mathcal{U}| \cdot \sigma \log n) = O(n^2 \log n)$ time.

Recursive BinaryPartition. The height of the recursive binary partition tree is at most the length of the longest MCS, which is at most $\min\{|X|, |Y|\} = O(n)$.

The first operation at each step consists in computing the set Ext_P: by scanning the unshiftable edges we can trivially find the cross in $O(|\mathcal{U}|) = O(n^2)$ time, and by another scan the mikado and Ext_P.

When it is nonempty, we check for maximality of P for each of its elements (recalling $|Ext_P| = O(n)$), which takes $O(n)$ time by employing Sakai's maximality test [11]. If the test is positive, we only need to perform a leftward re-map of the new edge, which can be done in logarithmic time, totalizing $O(|\mathcal{U}| + |Ext_P|(n + \log n)) = O(n^2)$ time.

Overall, the delay of BinaryPartition is the cost of a root-to-leaf path in the recursion tree, which has depth $\leq n$. This leads to a polynomial-delay algorithm with delay $O(n^3)$ and a polynomial-time preprocessing cost of $O(n^2 \log n)$. The space required is $O(n^2)$, as we need to store the set Ext_P for all recursive calls in a root-to-leaf path plus the set of unshiftable edges $\mathcal{U}$.

In the full version of the paper we provide a more refined algorithm which achieves $O(n \log n)$ delay, with the same preprocessing time and space. An ideal method would yield each distinct MCS in time proportional to its length: as the latter can be $\Theta(n)$, this would take linear time in n. In our refined algorithm we only spend a further logarithmic time factor per solution, so it is quite close to the ideal method. As for space and preprocessing time, the quadratic factor is unavoidable when employing the possibly quadratic unshiftable edges in $\mathcal{U}$.

4 Proof of Theorem 3

In this section we finalize the proof of Theorem 3, at the heart of our results. We introduce the concept of *certificate edges*, and use it to show sufficiency and necessity of the two conditions *(1)* and *(2)* in Theorem 3.

4.1 Certificate Edges

Certificate edges are defined as follows, and illustrated in Fig. 5.

Definition 9. *An edge* $(i', j') \in \mathcal{U}$ *is a* certificate *for another edge* (i, j) *if* $(i', j') \in G(\mathcal{I}_X(i), \mathcal{I}_Y(j))$ *and no* $(x, y) \in \mathcal{U} \setminus \{(i', j')\}$ *has* $x \in (i, i']$, $y \in (j, j']$. *In this case we say that* (i', j') certifies (i, j). *We denote with* $\mathcal{C}_{(i,j)}$ *the set of certificates of edge* (i, j). *An edge* $(i, j) \in \mathcal{U}$ *is called a* root *iff* $\mathcal{C}_{(i,j)} = \emptyset$.

Definition 10. *A* certificate mapping *is a mapping in which the rightmost edge is a root, and each edge except the leftmost is a certificate for the one to its left.*

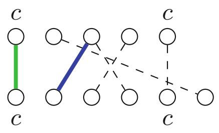


Fig. 5. The only certificate for the green edge is drawn in blue. (Color figure online)

Lemma 2. *Let $M = \{r_1, ..., r_N\}$ be a maximal certificate mapping in $G(X', Y')$ of a common subsequence $S = S_1 \cdots S_N$ between X' and Y', where $r_1 = (i_1, j_1)$.*

1. *M is a rightmost maximal mapping of unshiftable edges in $G(X', Y')$, and*
2. *if $G(X'_{<i_1}, Y'_{<j_1}) \cap \mathcal{U} = \emptyset$, then $M \in MCS(X', Y')$.* *(proof in full version)*

We now define the FIND$_R$ procedure, used to generate certificate mappings. This procedure implicitly finds the C from Theorem 2. Given an unshiftable edge, FIND$_R$ chooses one of its certificates and recurs until it gets to a root edge.

$$\text{FIND}_R(i, j) = \{(i, j)\} \cup \left(\cup_{(h,k) \in \mathcal{C}_{(i,j)}} \text{FIND}_R(h, k) \right)$$

Proposition 2. *Let (l, m) be any edge of the graph, and $(i, j) \in Ext_{(l,m)}$ in the set of extensions of (l, m). Then FIND$_R(i, j)$ returns a certificate mapping with first edge (i, j), such that the corresponding subsequence is $M \in MCS(X_{>l}, Y_{>m})$.*

Proof. The procedure FIND$_R(i, j)$ generates a certificate mapping starting with edge (i, j) by definition. Since $(i, j) \in Ext_{(l,m)}$, there cannot be any unshiftable edges in the subgraph $G(X[l+1, ..., i], Y[m+1, ..., j])$. By setting $X' = X_{>l}$ and $Y' = Y_{>m}$ in Lemma 2, $M \in MCS(X_{>l}, Y_{>m})$ and is rightmost. □

4.2 Necessary and Sufficient Conditions

Necessity. First of all, we will prove that conditions *(1)* and *(2)* of Theorem 3 are necessary. Let $P\,c$ be a valid prefix of some $W \in MCS(X, Y)$.

First, we show that condition *(1)* holds, namely, there exists $(i, j) \in Ext_P$ corresponding to character c. We use contradiction below, supposing that none of the edges in Ext_P correspond to c.

By Sakai's characterization of maximality, for all indices $k \leq |W|$ we have $X_k \cap Y_k = \emptyset$. Let $\hat{k} = |P|$, and thus $W = P\,W_{>\hat{k}}$ and $W_{>\hat{k}}$ starts with c because $P\,c$ is a valid prefix of W. By definition, $X_{\hat{k}} \cap Y_{\hat{k}} = \emptyset$, where $X_{\hat{k}}$ and $Y_{\hat{k}}$ are given by the parts of the strings between the leftmost mapping L_P of P and the rightmost mapping of $W_{>\hat{k}}$. The first edge of the latter mapping is $(i, j) \in \mathcal{U}$ corresponding to character c as $W_{>\hat{k}}$ starts with c. By contradiction, suppose $(i, j) \notin Ext_P$. We now have two cases.

Case $(i, j) \notin Mk_P$: this implies that $i > f_1$ or $j > e_2$, where f_1 and e_1 are those given in Definition 8. Therefore $X_{\hat{k}} \cap Y_{\hat{k}} \neq \emptyset$ as there would be at least the character corresponding respectively to f or e. This is a contradiction.

Case $(i, j) \in Mk_P \setminus Ext_P$: this implies that $\exists(h, k) \in Mk_P \setminus (i, j)$ such that $h \leq i$ and $k \leq j$. Then $X_{\hat{k}} \cap Y_{\hat{k}} \neq \emptyset$ as we would have edge (h, k) in $G(X_{\hat{k}}, Y_{\hat{k}})$, giving a contradiction.

Second, we prove the necessity of condition *(2)*, namely, $P \in MCS(X_{<i}, Y_{<j})$.

To this end, we need a brief remark on the restriction of maximals: let $W \in MCS(X, Y)$ and $\{(x_1, y_1), ..., (x_{|W|}, y_{|W|})\}$ any mapping spelling W in the two strings. Given any $k \leq |W|$, we have $W_{<k} \in MCS(X_{<x_k}, Y_{<y_k})$.

Let $P\,c$ be a valid prefix of some $W \in MCS(X, Y)$, and $\hat{k} = |P|$. In the first part of the proof we have shown that the first edge of the rightmost mapping of $W_{>\hat{k}}$ is some $(i, j) \in Ext_P$ corresponding to c. Therefore, let us consider the mapping for W consisting of P in leftmost form, and $W_{>\hat{k}}$ in rightmost form. Applying the above remark for $k = \hat{k}+1$ we get $W_{<\hat{k}+1} = P \in MCS(X_{<i}, Y_{<j})$.

Sufficiency. Suppose that conditions *(1)* and *(2)* of Theorem 3 hold. By Proposition 2, $\textsc{find}_R(i, j) = C \in MCS(X_{>l}, Y_{>m})$. Since $P \in MCS(X_{<i}, Y_{<j})$ by hypothesis, we have $P\,C \in MCS(X, Y)$ by Theorem 2. The latter string starts with $P\,c$, which is therefore a good prefix.

5 Conclusions and Acknowledgements

In this paper we have studied the Maximal Common Subsequences (MCSs), and investigated their combinatorial nature by familiarizing with some of their properties. Circumventing various pitfalls, we ultimately provided an efficient, binary partition-based, polynomial-delay algorithm for listing all MCSs on an equivalent bipartite graph problem.

The work done in this paper was partially funded by NII, Japan and JST CREST, Grant Number JPMJCR1401. The work was partially done at author TU's laboratory when author AC was a postdoc at NII, author RG was on leave from the University of Pisa to visit NII, and author GP was visiting NII.

References

1. Abboud, A., Backurs, A., Williams, V.V.: Tight hardness results for LCS and other sequence similarity measures. In: 2015 IEEE 56th Annual Symposium on Foundations of Computer Science, pp. 59–78. October 2015
2. Apostolico, A.: Improving the worst-case performance of the Hunt-Szymanski strategy for the longest common subsequence of two strings. Inf. Process. Lett. **23**(2), 63–69 (1986)
3. Bergroth, L., Hakonen, H., Raita, T.: A survey of longest common subsequence algorithms. In: Proceedings Seventh International Symposium on String Processing and Information Retrieval. SPIRE 2000, pp. 39–48. September 2000

4. Eppstein, D., Galil, Z., Giancarlo, R., Italiano, G.F.: Sparse dynamic programming I: linear cost functions. J. ACM **39**(3), 519–545 (1992)
5. Fraser, C.B., Irving, R.W., Middendorf, M.: Maximal common subsequences and minimal common supersequences. Inf. Comput. **124**(2), 145–153 (1996)
6. Hirschberg, D.S.: Algorithms for the longest common subsequence problem. J. ACM **24**(4), 664–675 (1977)
7. Hunt, J.W., Szymanski, T.G.: A fast algorithm for computing longest common subsequences. Commun. ACM **20**(5), 350–353 (1977)
8. Kanté, M.M., Limouzy, V., Mary, A., Nourine, L.: On the enumeration of minimal dominating sets and related notions. SIAM J. Discrete Math. **28**(4), 1916–1929 (2014)
9. Lawler, E.L., Lenstra, J.K., Rinnooy Kan, A.H.G.: Generating all maximal independent sets: NP-hardness and polynomial-time algorithms. SIAM Journal on Computing **9**(3), 558–565 (1980)
10. Masek, W.J., Paterson, M.S.: A faster algorithm computing string edit distances. J. Comput. Syst. Sci. **20**(1), 18–31 (1980)
11. Sakai, Y.: Maximal common subsequence algorithms. In: Navarro, G., Sankoff, D., Zhu, B. (eds.) Annual Symposium on Combinatorial Pattern Matching (CPM 2018), vol. 105, Leibniz International Proceedings in Informatics (LIPIcs), pp. 1:1–1:10. Dagstuhl, Germany, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2018)
12. Wagner, R.A., Fischer, M.J.: The string-to-string correction problem. J. ACM **21**(1), 168–173 (1974)