Algorithmica (2022) 84:757–783
https://doi.org/10.1007/s00453-021-00898-5

Enumeration of Maximal Common Subsequences Between Two Strings

Alessio Conte[1] · Roberto Grossi[1] · Giulia Punzi[1] · Takeaki Uno[2]

Received: 31 August 2020 / Accepted: 14 November 2021 / Published online: 12 January 2022

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.

Keywords Maximal common subsequence · Subsequence · LCS · Enumeration · Polynomial-delay

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

✉ Giulia Punzi
giulia.punzi@phd.unipi.it

Alessio Conte
conte@unipi.it

Roberto Grossi
grossi@unipi.it

Takeaki Uno
uno@nii.jp

1 University of Pisa, Pisa, Italy

2 National Institute of Informatics, Tokyo, Japan

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.

1.1 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 indices $0 \leq i_0 < \cdots < i_{|S|-1} < |X|$ 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 $X = \texttt{TCACAG}$ and $Y = \texttt{GTACTA}$, where $MCS(X, Y) = \{\texttt{TACA}, \texttt{G}\}$. A greedy left-to-right common sequence is not necessarily an MCS: reading X from left to right and keeping the left-most matching character in Y when possible gives $W = \texttt{TCA}$, which is not in $MCS(X, Y)$ as $\texttt{TCA} \subset \texttt{TACA}$.

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 such that $n = \max\{|X|, |Y|\}$ 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 $O(\sigma n \log n)$-delay enumeration algorithm for Problem 1, with $O(\sigma n^2 \log n)$ preprocessing time and $O(n^2)$ space.*

1.2 Motivation and Relation to Previous Work

Maximal common subsequences were first introduced in the mid-90s by Fraser et al. [9]. Here, the concept of MCS was a stepping stone for one of the main problems

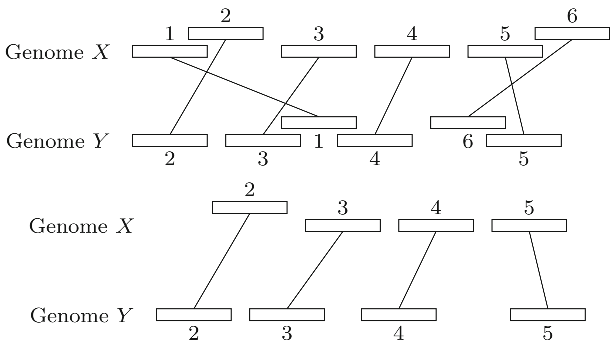


Fig. 1 On the top, a simplified representation of two genomes for whom 6 potential anchors have been identified. On the bottom, a possible colinear alignment of the anchors. A different colinear alignment would be given for instance by 2, 3, 4, 6. (Picture inspired by Fig. 3 from Delcher et al. [8].)

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 [4,10,18,22] have thoroughly been studied in a plethora of string comparison application domains, like spelling error correction, molecular biology, and plagiarism detection, to name a few, little is known for MCSs. In general, LCSs only provide us 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. Thus, MCS could in principle provide helpful information in all the applications where LCS are used.

Furthermore, a direct application for the different alignments highlighted by MCSs can be found in the field of comparative genomics. Because of the sheer size of the input, comparative genomics requires more specific tools than general string comparisons, thus branching into a field of its own. Some of the tools employed for genome comparison solve problems akin to the MCS problem, relying on *anchor-based* methods [5,8,16]. Given two genomes, these methods first look for potential *anchors*, which are long meaningful chunks of the genomes that are considered to be matching (see Fig. 1). Final anchors are extracted from these by computing a *colinear* (i.e. occurring in the same order in both genomes) sequence of non-overlapping potential anchors; eventual small gaps between the anchors are then filled separately. If we interpret the two sequences of potential anchors as strings, finding MCS between these helps find the optimal colinear alignment to later extract the anchors, and run the final alignment methods. An efficient algorithm to find MCS could therefore be employed to speed up these time-consuming comparison tools.

A drawback of LCS is that there is a quadratic conditional lower bound for their computation, based on the Strong Exponential Time Hypothesis [1].[1] On the other

[1] The Strong Exponential Time Hypothesis (SETH) [13] states that $\lim_{k\to\infty} s_k = 1$, where $s_k = \inf\{\delta \mid k\text{-SAT can be solved in } O(2^{\delta n}) \text{ time}\}$. It is widely believed to be true, and it has been used to prove conditional lower bounds for a variety of problems (see [2] for some examples).

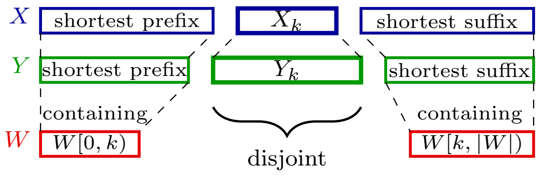


Fig. 2 Visual representation of Sakai's characterization

hand, no quadratic bound exists for MCS: in fact, a recent paper by Sakai [20] presents an algorithm that deterministically finds one MCS between two strings of length $O(n)$ in $O(n \log\log n)$ time, which he later improved to $O(n\sqrt{\log n / \log\log n})$ time in [21]. These algorithms can also be used to extend a given sequence to a maximal one in the same time. Furthermore, $O(n)$-time algorithms to check whether a given subsequence is maximal are described in the papers. To this end, in [20] the author gives a neat characterization of MCSs, which will be useful later, as stated in Lemma 1 and illustrated in Fig. 2.

Lemma 1 (MCS Characterization [20]) *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 for all $k \in [0, |W|]$, X_k and Y_k are disjoint (that is, they share no common characters).*

In what follows, we will showcase some pitfalls that arise when trying to trivially extend the aforementioned results to MCS enumeration. It is worth noting that even though there are a few more different approaches in the literature to find LCS or common subsequences with some kind of constraints (e.g. common subsequence trees [11], common subsequence automata [7]), the approaches described in this section are the only ones which directly deal with MCS. Thus, for space and clarity reasons, we will only focus on these latter approaches, and the pitfalls arising from their extensions. Still, to the best of our knowledge, other approaches in the literature do not immediately extend to efficient MCS enumeration either.

1.3 Strategic Pitfalls

The aforementioned results by Fraser et al. and Sakai 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 [9]: 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 $X = \mathtt{TAATAATAAT}$, $Z = \mathtt{TATATATATAT}$ 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 halfway we would compute $MCS(X, Z)$, which can be shown to have size $O(\exp(|X|))$. This means that it would require time exponential in the size of the input to provide just a single solution as output.

As for the approach in [20], 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 and Y_k are not disjoint, that is, they have some characters in common. Then, for these values, compute all distinct characters c which occur in both X_k and Y_k, and for each of these recur on the extended sequences $W' = W[0, \dots, k-1]\, c\, W[k, \dots, |W|-1]$. For instance, given the strings $X = \mathtt{ACACA}$ and $Y = \mathtt{ACACACA}$ with starting sequences $W = \mathtt{A}$ and $W = \mathtt{C}$, 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.

Lastly, the refined approach given in [21] also seems to not have an immediate extension to the enumeration of MCS. This latter approach builds a string W by going from left to right in the input strings, and adding common characters to W in a greedy way, like in Example 1. Once it gets to the end of the strings, it can identify whether a character is surely the last one for some MCS. Fixing this character as the last for W, it recursively looks for an MCS in the prefixes of the strings up to the last occurrences of the selected character, adding more letters to W whenever possible, to finally yield an MCS.

Example 4 Consider strings $X = \mathtt{GATAGAC}$ and $Y = \mathtt{AGATACAGA}$, with MCS given by $\{\mathtt{GATAGA}, \mathtt{GATAC}\}$. First note that, to adapt the algorithm to enumeration, we need to consider multiple starting characters, as otherwise we could miss some MCS in which the chosen starting letter does not appear. Reasonably, we choose all letters that do not have an insertion before them: in our example we would start with A and G. Further note that choosing different starting characters is not enough, as otherwise the algorithm would deterministically only find at most $|\Sigma|$ MCS. Thus, to perform enumeration based on this algorithm we need to choose all possible insertions, at every step, when going left to right. This is equivalent to a greedy left-to-right approach, which may yield the same sequence multiple times: in our example, string GATAGA is output 6 times, and string GATAC 3 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 [14], or would imply $P = NP$ [17]. As we will see, solving Problem 1 can lead to further pitfalls that we circumvent in this paper.

Our Approach. The approach that will finally lead us to our result is a *prefix-incremental* one: given P a prefix of an MCS, we will be able to identify all characters c such that $P\,c$ is still a prefix of some MCS. As we will see in the next section, this task is highly non-trivial, and leads to several more pitfalls of its own.

1.4 Overview

In the rest of the paper, we start by showing how to interpret the MCS problem as a graph problem, in Sect. 2. This change of perspective will lead us to our central combinatorial result, Theorem 2, which will be the basis of our prefix-incremental algorithm; its involved proof will be detailed in Sect. 3. The theorem will lead us naively into our first polynomial-delay enumeration algorithm, introduced in Sect. 4. Lastly, Sect. 5 will present a refinement of the initial algorithm, allowing us to finally reach the bounds described in Theorem 1.

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 to get some insight on how to combine MCS. Afterward, this characterization will be reformulated in an operative way, leading to an algorithm for MCS enumeration.

2.1 String Bipartite Graph

Definition 3 Given two strings X, Y, their *string bipartite graph* $G(X, Y)$ has vertex set $V = \{x_0, \ldots, x_{|X|-1}, y_0, \ldots, y_{|Y|-1}\}$, where x_i, y_j represent respectively the i-th position of X and the j-th position of Y, and edge set $E = \{(x_i, y_j) \mid X[i] = Y[j]\}$. Each edge, called *pairwise occurrence*, connects positions with the same character in different strings. Wherever it is clear from the context, we will denote edge (x_i, y_j) simply as (i, j).

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. A mapping that is inclusion-maximal is called a *maximal mapping*.

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 $X = \texttt{AGG}$ and $Y = \texttt{AGAG}$, with $MCS(X, Y) = \{\texttt{AGG}\}$. $G(X, Y)$ has an inclusion-maximal mapping corresponding to $\texttt{AG} \notin MCS(X, Y)$.

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.

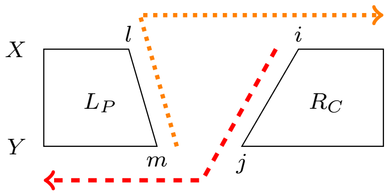


Fig. 3 For their concatenation to be an MCS, P has to be maximal in the red dashed part and C in the orange dotted one (color figure online)

Remark 1 Given any common subsequence $S = s_0 \cdots s_N$ of X, Y, we can always build its unique *rightmost mapping* as follows. Start at s_N, and let $i_N = \max\{i \mid X[i] = s_N\}$ and $j_N = \max\{j \mid Y[j] = s_N\}$; consider then for every $N > h \geq 0$, $i_h = \max\{i < i_{h+1} \mid X[i] = s_h\}$ and $j_h = \max\{j < j_{h+1} \mid Y[j] = s_h\}$. Edges $(i_0, j_0), \ldots, (i_N, j_N)$ form a rightmost mapping spelling S. Symmetrically, we can always build the unique *leftmost mapping* of S.

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'. This will help us develop a prefix-expanding strategy for $MCS(X\,X', Y\,Y')$, as we can consider $P \in MCS(X, Y)$ as a prefix, and the first characters of strings in $MCS(X', Y')$ as its extensions.

Remark 2 A simple concatenation of the pairwise MCS fails: consider for example $X =$ `AGA`, $X' =$ `TGA`, $Y =$ `TAG` and $Y' =$ `GAT`, with $MCS(X\,X', Y\,Y') =$ {`AGGA`, `AGAT`, `TGA`}. We have $MCS(X, Y) =$ {`AG`} and $MCS(X', Y') =$ {`GA`, `T`}. Combining the latter two sets we find the sequence `AGGA`, which is in fact maximal, but also `AGT`, which is not maximal as `AGT` $\subset$ `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 $i + 1$.

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. 3). Then*

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

Proof To ensure the equivalence, it is sufficient to show that Sakai's interspaces for string $P\,C$ 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 $P = p_1 \cdots p_s$, $C = c_1 \cdots c_r$, and let us study the k-th interspace for the subsequence $P\,C$.

Case $k < s$: the shortest suffixes of X, Y containing $p_{k+1}, \ldots, p_s, c_1, \ldots, c_r$ as a subsequence are unchanged from the shortest suffixes of $X_{<i}, Y_{<j}$ containing $p_{k+1}, \ldots, p_s$, since C is already in rightmost form starting exactly at (i, j). Let $X_{<u+1}, Y_{<v+1}$ be the shortest prefixes of X, Y containing $p_1, \ldots, p_k$ as a subsequence. Then, by definition of leftmost mapping, (u, v) must be the k-th edge of the leftmost mapping of P, which occurs before edge (i, j) by hypothesis. The shortest

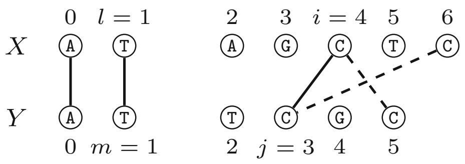


Fig. 4 The possibility for insertion is not sufficient to discard a character as a valid extension: edge (i, j) corresponds to valid character C even though both its endpoints can be re-mapped (dashed) to allow for insertions (see Remark 3)

prefixes of X, Y containing $p_1, \ldots, p_k$ are thus also unchanged from the ones for $X_{<i}, Y_{<j}$. 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 $X[l+1, \ldots, i-1]$ and $Y[m+1, \ldots, j-1]$. 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 $P\,C$. □

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 the 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 the following task without explicitly knowing C: given a valid prefix P with leftmost mapping ending at (l, m), find the edges (i, j) whose corresponding characters yield a valid prefix when used to extend P.

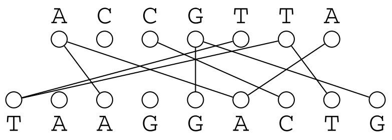
Remark 3 Note that, when looking for edge (i, j), the occurrence of an insertion when moving its endpoints is not enough to discard the candidate. Consider as an example $X =$ ATAGCTC and $Y =$ ATTCGC, with valid prefix $P =$ AT ending at $(l, m) = (1, 1)$. Consider edge $(i, j) = (4, 3)$; clearly $P \in MCS(X_{<i}, Y_{<j})$. Also, (i, j) can be moved in both strings to allow for insertions of characters G and T. Nonetheless, its corresponding character C still generates the valid prefix ATC. This is illustrated in Fig. 4. Along the same lines, Sakai's algorithm cannot help here. It generates an 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 unshiftable 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 5 Consider $X = \texttt{ACCGTTA}$ and $Y = \texttt{TAAGGACTG}$ with $MCS(X, Y) = \{\texttt{ACG}, \texttt{ACT}, \texttt{AGA}, \texttt{AGT}, \texttt{TA}, \texttt{TT}\}$. The unshiftable edges for these two strings are the following ones:



A symmetric definition of *left-unshiftable* edges could be given by considering maximal leftmost mappings, and these are related to *k-dominant edges*, a concept introduced in 1985 by Apostolico [3]: k-dominant edges turn out to be a subset of left-unshiftable edges.[2]

Unshiftable edges can also be characterized more directly, as stated in Proposition 1.

Proposition 1 *An edge* (i, j) *is unshiftable if and only if either* (1) *it corresponds to the rightmost pairwise occurrence of* $X[i] = Y[j]$ *in the strings, or* (2) *there is at least one unshiftable edge in the subgraph* $G(\mathcal{I}_X(i), \mathcal{I}_Y(j))$.

Proof ($\Longrightarrow$) Let $(i, j) \in \mathcal{U}$, then by definition there exists a maximal rightmost mapping $\mathcal{R} = r_1, \ldots, r_N$ such that $(i, j) = r_p$ for some $1 \leq p \leq N$. If $p = N$, by definition of rightmost mapping, r_N corresponds to the last pairwise occurrence of some character, thus it satisfies (1). Consider now $p < N$, and let r_p correspond to some character c. By definition of unshiftability, $r_{p+1} \in \mathcal{U}$. By definition of rightmost maximal mapping, 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 $G(\mathcal{I}_X(i), \mathcal{I}_Y(j))$.
($\Longleftarrow$) 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. □

An immediate consequence of Proposition 1 is the following fact, which will give us an operative way of finding the set of unshiftable edges:

[2] The author employs these edges to improve the Hunt–Szymansky algorithm [12], which extracts one LCS of two strings of length n in $O((r + n) \log n)$, where r is the total number of ordered pairs of positions at which the two sequences match, that is, the number of edges in the string bipartite graph.

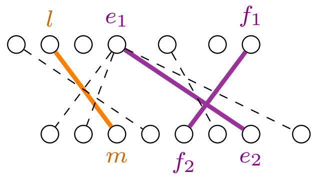


Fig. 5 Graphical representation of the cross $\langle e, f\rangle$ for edge (l, m), drawn in purple: $e = (e_1, e_2)$, $f = (f_1, f_2)$ are the first unshiftable edges soon after (l, m) (color figure online)

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.*

Remark 4 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 $X = \texttt{AAGAAG}$ and $Y = \texttt{AAGA}$. In $G(X, Y)$ we have a maximal rightmost mapping for $\texttt{AAG}$, but $\texttt{AAG} \subset \texttt{AAGA} \in MCS(X, Y)$.

Remark 5 Unshiftable edges can be dense in $G(X, Y)$. For example, consider $X = \texttt{A}^n(\texttt{CA})^n$ and $Y = \texttt{A}^n\texttt{C}^n$: every $\texttt{A}$ of Y has out-degree of unshiftable edges equal to the number of $\texttt{C}$s in X, that is $O(n)$. The total number of unshiftable edges is therefore $|\mathcal{U}| = O(n \cdot n) = O(n^2)$.

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 C's 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. 5) is given by (at most) two unshiftable edges $e = (e_1, e_2)$, $f = (f_1, f_2)$ such that

$$e_1 = \min\{h_1 > l \mid \exists h_2 > m : (h_1, h_2) \in \mathcal{U}\} \text{ and } e_2 = \min\{h_2 > m : (e_1, h_2) \in \mathcal{U}\},$$
$$f_2 = \min\{h_2 > m \mid \exists h_1 > l : (h_1, h_2) \in \mathcal{U}\} \text{ and } f_1 = \min\{h_1 > l : (h_1, f_2) \in \mathcal{U}\}.$$

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 $G(X[e_1, \ldots, f_1], Y[f_2, \ldots, e_2])$,

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

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

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

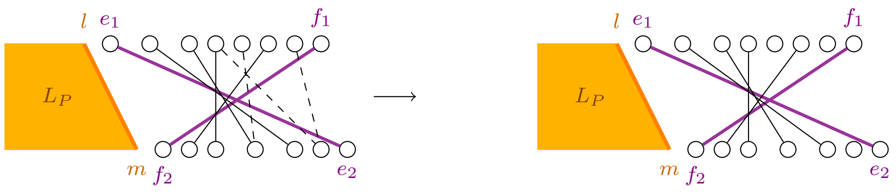


Fig. 6 Extraction of $Ext_{(l,m)}$ from the set $Mk_{(l,m)}$, pictured on the left. The edges belonging to $Mk_{(l,m)} \setminus Ext_{(l,m)}$ are dashed

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

Definitions 7 and 8 find their application in identifying a valid prefix extension, as shown in Fig. 6 and discussed next.

2.4 Valid Prefix Extensions

Let P be a valid prefix with leftmost mapping L_P ending with the 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. 6.

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 with the 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. 3. 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 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.

3.1 Certificate Edges

Given an edge, we call its certificate edges the unshiftable edges that occur right after the edge in question and before the next pairwise occurrences of the corresponding character, with no other unshiftable in between.

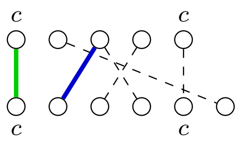


Fig. 7 The only certificate for the green bold edge corresponding to character c is drawn in solid blue (all dashed edges are unshiftable)

Recalling that $\mathcal{I}_X(i) = X[i+1, \ldots, next_X(i)]$, certificate edges are defined as follows, and illustrated in Fig. 7.

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 $(x, y) \in \mathcal{U} \backslash \{(i', j')\}$ 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 $\mathcal{C}_{(i,j)} = \emptyset$.

Certificates are given this name as they certify the unshiftability of the edge, as detailed in the following:

Remark 6 Note that if $\mathcal{C}_{(i,j)} \neq \emptyset$, then $(i, j) \in \mathcal{U}$. In fact, let $(i', j') \in \mathcal{C}_{(i,j)}$; then, by definition of certificate, (i, j) are the rightmost occurrences of its corresponding character c in $X_{<i'}$ and $Y_{<j'}$. Thus, by Fact 1, $(i, j) \in \mathcal{U}$.

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.

Using certificate edges, we can prove the following sufficient condition that will help us identify MCSs.

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

1. *M is a rightmost maximal mapping of unshiftable edges in $G(X', Y')$, and*
2. *if $G(X'_{\leq i_1}, Y'_{\leq j_1}) \cap \mathcal{U} = \{r_1\}$, then $M \in MCS(X', Y')$.*

Proof 1. By contradiction, assume that M is not rightmost. This means that there exists at least one edge r_i, corresponding to some $c \in \Sigma$, such that c occurs between r_i and r_{i+1} in either one of X' or Y'. Therefore, r_{i+1} is not a certificate of r_i: contradiction.

2. Assume that $G(X'_{\leq i_1}, Y'_{\leq j_1}) \cap \mathcal{U} = \{r_1\}$. By contradiction, let M not be maximal: let k be the minimum index such that we can insert a character c between S_k and S_{k+1}. If $k = 0$ (insertion at beginning), then there is a pairwise occurrence of c before r_1; let (i, j) be the rightmost such pairwise occurrence. Then, r_1 is a certificate for (i, j), and therefore by Remark 6, $(i, j) \in G(X'_{\leq i_1}, Y'_{\leq j_1}) \cap \mathcal{U}$: contradiction. We can therefore suppose $k > 0$. If we can insert c between S_k and S_{k+1}, it means that the k-th interspace is non-empty and contains c. This is equivalent to saying that we can find an edge corresponding to character c in the substrings between the leftmost mapping of S_k and the rightmost mapping of S_{k+1}. Since the latter is already in rightmost form, we will perform a re-mapping of just S_k. First, note that when remapping S_k in any way, we can only shift one

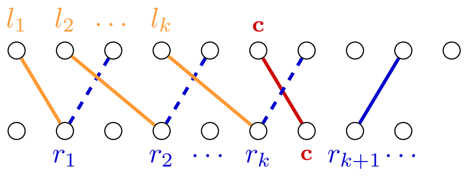


Fig. 8 The red edge corresponding to character c is the insertion that takes place between edges l_k and r_{k+1}. Solid orange edges $l_1, \ldots, l_k$ are the minimal leftward re-map of edges $r_1, \ldots, r_k$, shown dashed in blue (color figure online)

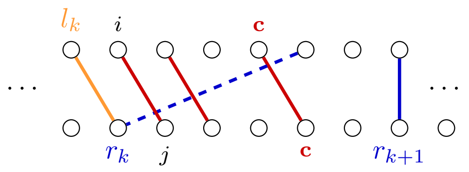


Fig. 9 Without loss of generality, we can consider the leftmost insertion given by edge (i, j)

endpoint per edge. This is because shifting both endpoints would mean that we have a character match, and therefore an insertion , which is impossible before the k-th edge by hypothesis. Let $l_1, \ldots, l_k$ be the *minimal* leftward re-map of the edges $r_1, \ldots, r_k$ that allows for the insertion of c between l_k and r_{k+1}. That is, the endpoints of $l_1, \ldots, l_k$ are the rightmost that allow for insertion. Note that in such a minimal re-map, the edges' endpoints that get re-mapped always belong to the same string, as otherwise we would either have crossing edges, and thus not a valid mapping, or have a superfluous re-map, contradicting minimality. Such a re-map is illustrated in Fig. 8.

Let (i, j) corresponding to some character c be the rightmost insertion that can occur between edge l_k and edge r_{k+1}. Then, (i, j) is an unshiftable edge, since it has edge r_{k+1} as a certificate. It is possible that we can insert a string of characters ending with c between l_k and r_{k+1} , as illustrated in Fig. 9. In this case, let us choose for each of these characters the edge corresponding to its rightmost pairwise occurrence before the edge for the next character. All of these edges are unshiftable since each one is a certificate for the previous, starting with the one for c. The following proof can proceed considering without loss of generality the leftmost of these insertions as edge (i, j).

By choosing the edge in this fashion, we guarantee that there aren't any more unshiftable edges (or even regular edges) between l_k and (i, j). Furthermore, since l_k was the minimal leftward re-map, there are no occurrences of its corresponding character between itself and (i, j), and so we find that l_k is certified by (i, j). Edge l_k is then also unshiftable. This now propagates backward for every h: let l_{h-1} correspond to character a, and assume that l_h is unshiftable. Then, there can be no occurrence of a between l_{h-1} and l_h: (1) there are no occurrences of a between l_{h-1} and r_{h-1} because the re-map was minimal; (2) there can be no occurrence of a between r_{h-1} and r_h because they belonged to a rightmost mapping (see Fig. 10).

Thus, l_{h-1} is unshiftable. Once we get to l_1, we obtain an unshiftable edge different from r_1 in the subgraph $G(X'_{\leq i_1}, Y'_{\leq j_1})$: contradiction. □

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$_R$

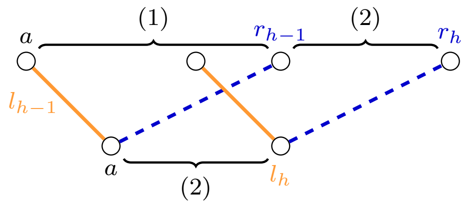


Fig. 10 Propagation of unshiftability for l_{h-1}. Edges l_{h-1}, l_h are shown in solid orange, while their rightmost counterparts r_{h-1}, r_h are dashed blue. Edge l_{h-1} corresponds to character a, which cannot appear in either (1) or (2): the former because of the minimality of the re-mapping, the latter because r_{h-1} is rightmost in the original mapping (color figure online)

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* $\text{FIND}_R(i, j)$ *returns a certificate mapping having as first edge* (i, j)*, such that the corresponding subsequence is* $M \in MCS(X_{>l}, Y_{>m})$*.*

Proof The procedure $\text{FIND}_R(i, j)$ 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 $G(X[l+1, \dots, i], Y[m+1, \dots, j])$, except for (i, j) itself. By setting $X' = X_{>l}$ and $Y' = Y_{>m}$ in Lemma 2, $M \in MCS(X_{>l}, Y_{>m})$ and is rightmost. □

3.2 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 do so by supposing that none of the edges in Ext_P correspond to c, and showing that this leads to a contradiction. By Sakai's characterization of maximality, for all indices $k \le |W|$ we have that X_k and Y_k are disjoint, that is, they share no common characters. 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, $X_{\hat{k}}$ and $Y_{\hat{k}}$ are disjoint, 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 $X_{\hat{k}}$ and $Y_{\hat{k}}$ cannot be disjoint, as there would be at least the character corresponding respectively to f or e. This is a contradiction.
- Case $(i, j) \in Mk_P \backslash Ext_P$: this implies that $\exists (h, k) \in Mk_P \backslash (i, j)$ such that $h \le i$ and $k \le j$. Then $X_{\hat{k}}$ and $Y_{\hat{k}}$ are not disjoint, as we would have the 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 $\{(x_1, y_1), \ldots, (x_{|W|}, y_{|W|})\}$ 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, $\text{FIND}_R(i, j) = C \in MCS(X_{>l}, Y_{>m})$. 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.

4 Baseline Algorithm for Polynomial-Delay MCS Enumeration

The characterization given in Theorem 3 immediately gives the "prefix-expanding" enumeration Algorithm 1, which progressively augments prefixes with characters that keep them valid, until whole MCSs are recursively generated. In this section, we will describe this baseline algorithm, and prove that it has polynomial-delay. It is worth noting that Theorem 3 guarantees that each recursive call yields at least one MCS; moreover, all the MCSs are listed once as different recursive calls originating from the same prefix P are performed by appending distinct characters.

Algorithm overview. Algorithm 1 employs a binary partition scheme.[3] First, it builds the necessary data structures (Sect. 4.1), and it finds the set of unshiftable edges in a polynomial preprocessing phase, using FINDUNSHIFTABLES as described in detail in Sect. 4.2. 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 over characters $c \in \Sigma$ such that $P\,c$ is valid, which is ensured by checking the two conditions of Theorem 3.

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 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 $c \in \Sigma$ the procedure loops over the edges of Ext_P corresponding to c (i.e. condition (1) of Theorem 3), to check whether any of these edges satisfy condition *(2)* of Theorem 3. If the check has positive answer for one of the edges, it means that $P\,c$ is a valid prefix: 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

[3] To be precise, each recursive node has up to σ child nodes; the binary partition is seen by the fact that each recursive call corresponds to "using the edge (l_c, m_c)" and the continuation after backtracking corresponds to "not using the edge (l_c, m_c)".

Algorithm 1 (Polynomial-Delay Enumeration Algorithm for MCS)

```
    Input: two strings $X, Y$ over given alphabet $\Sigma$
    Output: $MCS(X, Y)$
 1: procedure EnumerateMCS($X, Y, \Sigma$)
 2:     $\mathcal{U}$ = FindUnshiftables($(|X|, |Y|)$)
 3:     BinaryPartition(#, $\{(-1, -1)\}$)
 4: end procedure

    Input: a valid prefix $P$ and its leftmost mapping $L_P$
    Output: all strings in $MCS(X, Y)$ having $P$ as prefix
 5: procedure BinaryPartition($P, L_P$)
 6:     compute the set of extensions $Ext_P$ using $\mathcal{U}$
 7:     if $Ext_P = \emptyset$ then output $P$
 8:     else
 9:         for $c \in \Sigma$ do
10:             for $(i, j) \in Ext_P$ corresponding to $c$ do
11:                 if $P \in MCS(X_{<i}, Y_{<j})$ then
12:                     let $(l, m)$ be the last edge of $L_P$
13:                     find leftmost mapping of edge $(l_c, m_c)$ for $c$ in $G(X_{>l}, Y_{>m})$
14:                     BinaryPartition($P\,c, L_P \cup (l_c, m_c)$)
15:                     break              ▷ Exit the for loop of line 11, as it has already recurred with $c$
16:                 end if
17:             end for
18:         end for
19:     end if
20: end procedure

    Input: an edge $(i, j)$ of a string bipartite graph of two strings $X, Y$
    Output: set of unshiftable edges of $X_{<i}, Y_{<j}$
21: procedure FindUnshiftables($(i, j)$)
22:     $\mathcal{U} = \emptyset$
23:     for $c \in \Sigma$ do
24:         $l_X \leftarrow$ the rightmost occurrence of $c$ in $X_{<i}$
25:         $l_Y \leftarrow$ the rightmost occurrence of $c$ in $Y_{<j}$
26:         if $l_X \neq -1$ and $l_Y \neq -1$ and $(l_X, l_Y) \notin \mathcal{U}$ then
27:             add $(l_X, l_Y)$ to $\mathcal{U}$
28:             $\mathcal{U} = \mathcal{U} \cup$ FindUnshiftables($(l_X, l_Y)$)
29:         end if
30:     end for
31:     output $\mathcal{U}$
32: end procedure
```

$P\,c$ and $L_{P\,c}$. After the recursive call, the next character needs to be considered, and thus the algorithms breaks from the loop of line 10. The correctness of Algorithm 1 immediately follows from Theorem 3.

Polynomial-delay complexity. Recall that unshiftable edges are polynomial (quadratic) in size (Remark 5). We will show that their extraction in the preprocessing phase, as well as the computation of the Ext_P set at line 6 are polynomial-time. Then, the algorithm loops over each character of the alphabet, and checks whether the edges in Ext corresponding to it satisfy condition (2) of Theorem 3 (line 11). For a given edge, this check is also polynomial-time, as we could e.g. employ Sakai's algorithm [21]. Before recurring, the algorithm then updates the leftmost mapping by adding the new edge, which is once again a polynomial-time operation. Since every operation is polynomial-time, and every loop is over polynomial-size sets, each recursive call requires polynomial time. The working space is also polynomial. As for the final delay complexity, note that the height of the recursive binary partition tree is at most the length of the longest MCS, which is bounded by n, and that each leaf corresponds to a distinct solution. The delay of BinaryPartition is the cost of a root-to-leaf path in the recursion tree; since every step is polynomial-time, and the height of the tree is bounded by n, we obtain the following for Algorithm 1.

Theorem 4 *There is a polynomial-delay algorithm for Problem 1, which has polynomial-time preprocessing and uses polynomial space.*

In the rest of the section, we will describe the preprocessing phase and present the data structures in detail, in order to analyse the complexity of the baseline Algorithm 1. Specifically, we will prove.

Theorem 5 *There is a $O(\sigma n^3)$-delay algorithm for Problem 1, with $O(\sigma n^2 \log n)$ preprocessing time and $O(n^2)$ space.*

4.1 Data Structures

We start by describing the data structures that will be employed by Algorithm 1, allowing for fast queries over the strings X, Y and over the unshiftable edges set $\mathcal{U}$.

String data structures. The main operation that we perform multiple times during the execution of the algorithm is the computation of previous and next occurrences of a given character in the two strings X and Y. To be able to perform these operations in constant time, for each string we employ σ bitarrays of the same length as the string. These arrays $B_1^{(X)}, \dots, B_\sigma^{(X)}$ (analogously for Y) are such that

$$B_i^{(X)}[j] = 1 \iff X[j] = c_i; \quad B_i^{(Y)}[j] = 1 \iff Y[j] = c_i.$$

On these bitarrays we can implement rank and select directories in a succinct way: we need only $n + o(n)$ bits per array to do so, where n is the length of the array. With these directories, rank and select operations can be performed in $O(1)$ time [19]. That is, we can retrieve next and previous occurrences of any characters with respect to any positions of the strings in $O(1)$. The total time and space required to build these structures is $O(n\sigma)$.

Unshiftable data structures. For subsequent operations, we need to also store the unshiftable edges in different data structures, specifically in two arrays of arrays uX and uY. Each of these arrays gives priority to one of the two strings. Entry $uX[i]$ stores unshiftable edges (*wedges*) $(i, j_1), \dots, (i, j_s)$ as a sorted array of Y values $j_1, \dots, j_s$ as shown in Fig. 11. uY is built symmetrically with respect to Y.

To build these arrays we will consider the following strict orders

$$(i, j) <_{uX} (h, k) \iff i < h \text{ or } i = h \text{ and } j < k;$$
$$(i, j) <_{uY} (h, k) \iff j < k \text{ or } j = k \text{ and } i < j.$$

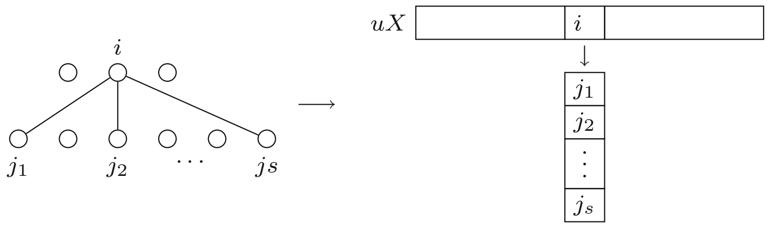


Fig. 11 For each node of X, wedges are sorted arrays of positions of Y

Given these arrays, we can check whether an edge (i, j) is unshiftable in $O(\log n)$ time by performing a binary search for j (or i) in the sorted array $uX[i]$ (or $uY[j]$), since $|uX[i]|, |uY[j]| \leq n$. With the same procedure, we can also find, given i and j', the first unshiftable (i, j) such that $j > j'$. These arrays are constructed and filled during the FindUnshiftables procedure, which we will describe next.

4.2 Finding Unshiftable Edges

During the preprocessing phase of the algorithm, we initialize the data structures and we compute the set of unshiftable edges with FindUnshiftables. We compute unshiftable edges by going backwards in the strings X and Y, exploiting Fact 1, which states that if (i', j') corresponds to the rightmost occurrences of a character c right before an unshiftable edge, then (i', j') is also unshiftable. We can thus start from the last pairwise occurrences of every character, then for every unshiftable edge (i, j) already found and for every character c, we can compute the rightmost occurrences of c in $X_{<i}$ and $Y_{<j}$, and mark the corresponding edge as unshiftable. Once we get to the beginning of the strings, we have correctly identified all unshiftable edges.

Analogously as we do for BinaryPartition, 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 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. In fact, 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 correctly marked as unshiftable.

Auxiliary data structures and complexity. During the FindUnshiftables procedure, we will need some intermediate support data structures to quickly check for membership and perform insertions. Since $\mathcal{U}$ is static after its creation, these data structures are only needed during the preprocessing phase: at the end of the procedure, they will be used to build the final arrays uX and uY, and then they will be deleted. First of all, in the FindUnshiftables procedure we need to ensure that no edge is visited twice. To this end, we employ two self-balancing binary search trees T_X, T_Y (for example, AVL or red-black trees), respectively storing the unshiftable edges with respect to orders $<_{uX}$, $<_{uY}$. Such trees allow for membership testing and insertion in $O(\log |\mathcal{U}|)$ time, and require $O(|\mathcal{U}|)$ space [15, Section 6.2.3]. The procedure considers each unshiftable edge exactly once. For each edge it goes through every character and finds its previous pairwise occurrences; then it checks for membership in the trees, and performs insertion if the edge is not found (see Line 26). Note that edge insertions are always performed for both trees. With the described data structures, looking for pairwise occurrences of the character requires constant time (using the string data structure), and membership tests and insertion for edges are performed in logarithmic time, leading to $O(\sigma |\mathcal{U}| \log |\mathcal{U}|)$ total time and $O(|\mathcal{U}|)$ space. Once the whole strings have been processed, each tree contains a sorted copy of the unshiftable edges, with respect to the corresponding order. Arrays uX, uY are created, and they can be filled with one more pass of the already sorted trees.

Recalling Remark 5, we have $|\mathcal{U}| = O(n^2)$, and thus the FINDUNSHIFTABLE procedure can be performed in $O(\sigma |\mathcal{U}| \log |\mathcal{U}| + |\mathcal{U}|) = O(\sigma n^2 \log n)$ time, and $O(|\mathcal{U}|) = O(n^2)$ space in the worst case.

4.3 Complexity Analysis

We now show that Algorithm 1 satisfies the complexity bounds of Theorem 5.

As detailed in Sects. 4.1–4.2, the preprocessing phase, which encompasses building the data structures and finding the set of unshiftable edges, can be performed in $O(\sigma n^2 \log n + n\sigma) = O(\sigma n^2 \log n)$ time and quadratic space.

Let us now study the complexity of a recursive call of the BINARYPARTITION procedure. 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 we find the mikado and Ext_P set. When it is nonempty, we loop over every character, finding its corresponding edges of Ext_P; for every such edge we perform the maximality check for P, which takes $O(n)$ time by employing Sakai's maximality test [20]. If the test is positive, we only need to perform a leftward re-map of the new edge, which can be done in constant time using the string data structure, and we move on to the next character. Thus, recalling that $|Ext_P| = O(n)$, the total time for one recursive call amounts to $O(|\mathcal{U}| + \sigma |Ext_P| \cdot n) = O(\sigma n^2)$ time.

Overall, the delay of BINARYPARTITION is given by the cost of a recursive call, times the height of the recursion tree. This leads to a polynomial-delay algorithm with delay $O(\sigma n^3)$ and a polynomial-time preprocessing cost of $O(\sigma n^2 \log n)$. 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 $\mathcal{U}$. Theorem 5 is thus proved.

While this is sufficient to prove the main result of the paper, i.e., that there exists a polynomial-delay and polynomial-space algorithm for the MCS enumeration problem, in the next section we focus on how to further improve its performance.

5 Improving the Delay in the Baseline Algorithm

In this section, we will describe a refinement of Algorithm 1, allowing us to achieve the bounds given in Theorem 1:

An ideal method would yield each distinct MCS in time proportional to its length: as the latter can be $\Theta(n)$, this would take time linear in n. In our refined algorithm we only spend a further logarithmic time factor per solution when the alphabet has constant size, so it is quite close to the ideal method in that instance. As for space and preprocessing time, the quadratic factor is unavoidable when employing the possibly quadratic unshiftable edges in $\mathcal{U}$.

There are two main refinements to the original algorithm: first, we will show how to quickly extract the Ext set at line 6 in time $O(\sigma \log n)$, instead of the original $O(n^2)$; then, we will provide a way to perform the maximality check of line 11 in constant time, without needing to employ the linear algorithm by Sakai.

5.1 Computing the Candidates for Extensions

We start by presenting a faster way to find characters corresponding to edges in Ext_P. This refinement will allow us to extract the Ext_P set in logarithmic time, speeding up line 6 of Algorithm 1. In what follows, let $first_S(c)$ and $last_S(c)$ be, respectively, the indices corresponding to the first and last occurrences of *character* c in the string S.

Remark 7 Let (l, m) be the last edge of the leftmost mapping for P. Then, for each edge $(h, k) \in Ext_P$ corresponding to some character c, either $h = first_{X_{>l}}(c)$ or $k = first_{Y_{>m}}(c)$. That is, at least one of the endpoints of (h, k) is the first occurrence of c after the leftmost mapping. This follows directly from the definition of the Ext_P set.

Given a leftmost mapping for P ending with the edge (l, m), for every $c \in \Sigma$ consider the two edges

$$u_c = (ix, jx) \text{ such that } \begin{cases} ix = first_{X_{>l}}(c) \\ jx = \min\{j > m \mid (ix, j) \in \mathcal{U}\} \end{cases}$$

$$v_c = (iy, jy) \text{ such that } \begin{cases} iy = \min\{i > l \mid (i, jy) \in \mathcal{U}\} \\ jy = first_{Y_{>m}}(c). \end{cases}$$

These two edges, represented in Fig. 12, are the first unshiftable edges completely after (l, m) that stem from the first occurrences of c respectively in X and Y after edge (l, m). Note that we can have $u_c = v_c$.

At this point, let $\mathcal{A} = \{u_c, v_c \mid c \in \Sigma\}$ be the set of all such edges, and consider its subset given by

$$\mathcal{E} = \{(i, j) \in \mathcal{A} \mid \nexists (h, k) \in \mathcal{A} \setminus (i, j) \text{ with } l < h \le i \text{ and } m < k \le j\}.$$

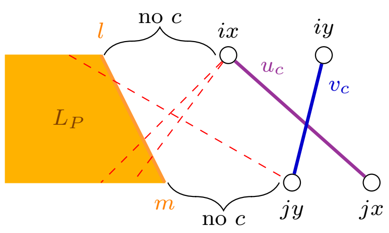


Fig. 12 Graphical representation of the edges u_c, v_c, shown respectively in bold purple and blue. The red dashed edges are other unshiftable edges stemming from ix, jy that are discarded since their other endpoint is out of bounds (color figure online)

Algorithm 2 (Finding the Extension Set of a Mapping)

```
    Input: the last edge $(l, m)$ of the leftmost mapping of a prefix $P$
    Output: $Ext_P$
 1: procedure FindExtensions($(l, m)$)
 2:     $\mathcal{E} = \emptyset$
 3:     for $c \in \Sigma$ do
 4:         let $ix = first_{X_{>l}(c)}, jy = first_{Y_{>m}}(c)$
 5:         $jx \leftarrow \min\{j > m \mid (ix, j) \in \mathcal{U}\}$
 6:         $iy \leftarrow \min\{i > l \mid (i, jy) \in \mathcal{U}\}$
 7:         add $(ix, jx), (iy, jy)$ to $\mathcal{E}$
 8:     end for
 9:     remove from $\mathcal{E}$ edges that have another element completely before them
10:     output $\mathcal{E}$
11: end procedure
```

As in the extraction of Ext from Mk, we have removed from $\mathcal{A}$ the edges that have another unshiftable completely before them. Note that $|\mathcal{E}| \leq |\mathcal{A}| \leq 2\sigma$, by definition.

Proposition 3

$$\mathcal{E} = Ext_P$$

Proof We have seen in Remark 7 that $Ext_P \subseteq \mathcal{A}$. By definition of Ext_P, in the reduction from $\mathcal{A}$ to $\mathcal{E}$ none of its edges will be removed. Therefore, $Ext_P \subset \mathcal{E}$. Specifically, edges $\chi_{(l,m)} = \langle e, f \rangle$ are in $\mathcal{E}$. Let us now consider the other inclusion. By contradiction, let $(i, j) \in \mathcal{E} \backslash Ext_P$. That is, either $(i, j) \notin Mk_P$, or there is an unshiftable edge completely between the leftmost mapping for P and (i, j). The latter condition is impossible by definition of $\mathcal{E}$, therefore $(i, j) \notin Mk_P$. This means that either $i > f_1$ or $j > e_2$. Both of these immediately lead to a contradiction: one of the edges $f \in \mathcal{A}$ or $e \in \mathcal{A}$ is completely before edge $(i, j) \in \mathcal{E}$. □

From this characterization, we have a simple procedure FindExtensions (Algorithm 2) that finds all candidates for extension. We restrict ourselves to $X_{>l}, Y_{>m}$, and for every character we find its first occurrences in both strings, which correspond to ix, jy. For every pair of occurrences, we compute the corresponding jx, iy by looking at the leftmost unshiftable edges in $X_{>l}, Y_{>m}$ which stem respectively from ix, jy. We then refine the set of edges we found by removing the ones that are preceded by another element of the set. The edges we obtain at the end of these steps are the Ext set.

Complexity analysis. In Algorithm 2, we first compute $u_c = (ix, jx)$, $v_c = (iy, jy)$ for every character c, inserting them into a list $\mathcal{A}$. To compute ix and jy we need to find the first occurrences of c in $X_{>l}$ and $Y_{>m}$, respectively, which can be done in constant time using the string data structure. Computing jx (resp. iy) is a bit more involved, as we need to find the leftmost unshiftable edge stemming from ix (resp. jy) and falling to the right of m (resp. l). To this end, we employ the unshiftable edges' arrays uX, uY: we look for (ix, j) with j greater than m (and the symmetric for jy), which can be done in $O(\log n)$. The worst-case total time for these steps is therefore $O(\sigma \log n)$. The only thing left to do to have our candidate edges is to extract $\mathcal{E}$ from $\mathcal{A}$. We first sort the list $\mathcal{A}$ according to $<_{uX}$, which takes $O(|\mathcal{A}| \log |\mathcal{A}|)$ time, e.g., using merge sort. At this point, we only need to go through the sorted elements once: for each $(i, j) <_{uX} (h, k)$ we discard (h, k) if $j < k$. In this way, we remove the unwanted edges in $O(|\mathcal{A}|)$ time.

Since $|\mathcal{A}| = O(\sigma)$, we have a time complexity of $O(\sigma \log \sigma)$ for the extraction of $\mathcal{E}$. Thus, when using procedure FindExtensions at line 6 instead of the naive computation, the time required for finding candidates for extension is improved from $O(n^2)$ to

$$O(\sigma \log n + \sigma \log \sigma) = O(\sigma \log n).$$

5.2 Maximality Check with Swings

The other major refinement of the algorithm concerns the maximality check $P \in MCS(X_{<i}, Y_{<j})$ at line 11 of Algorithm 1. Instead of using Sakai's method every time, we want to carry on information that allows us to decide immediately whether an extension is valid. To this end we now introduce the notion of *swing*.

Definition 11 Let L be a leftmost mapping for some string P, ending with edge (l, m). The *swing* of mapping L is a pair of integers $\ltimes(L) = (\ltimes_T(L), \ltimes_B(L))$ called respectively *top* and *bottom* swings, given by:

$$\ltimes_T(L) = \min\{i > l \mid P \notin MCS(X_{\leq i}, Y_{\leq m})\};$$
$$\ltimes_B(L) = \min\{j > m \mid P \notin MCS(X_{\leq l}, Y_{\leq j})\}.$$

Equivalently, $\ltimes_T(L) = \max\{i > l \mid P \in MCS(X_{<i}, Y_{\leq m})\}$, $\ltimes_B(L) = \max\{j > m \mid P \in MCS(X_{\leq l}, Y_{<j})\}$. Simply put, swings indicate how much we are allowed to move our edge while guaranteeing no insertions in the previous parts of the strings. Figure 13 gives a visual representation of the concept of swings.

With this notion, the maximality check becomes immediate:

Lemma 3 *Let P be a valid prefix, L_P its leftmost mapping, ending with the edge (l, m), and let $(\ltimes_T(L_P), \ltimes_B(L_P))$ be its swings. Furthermore, let $(i, j) \in Ext_P$ be a candidate edge. Then:*

$$P \in MCS(X_{<i}, Y_{<j}) \iff i \leq \ltimes_T(L_P) \text{ and } j \leq \ltimes_B(L_P). \quad (1)$$

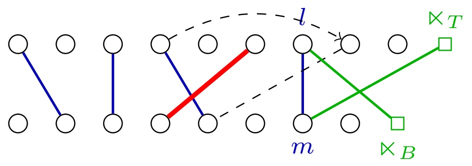


Fig. 13 The swings for the blue mapping are drawn in green; the thick red edge is the insertion resulting from the top swing; more specifically, it is the insertion possible if the character corresponding to edge (l, m) is re-mapped into the green top swing and the previous blue edge is re-mapped into the dashed edge (color figure online)

Proof If $P \in MCS(X_{<i}, Y_{<j})$, then in particular $P \in MCS(X_{<i}, Y_{\leq m})$ as $m < j$, and thus the swing $\ltimes_B(L_P)$ must occur at a position greater than j by Definition 11; symmetrically $P \in MCS(X_{\leq l}, Y_{<j})$ as $l < i$, and the swing $\ltimes_T(L_P)$ must occur at positions greater than or equal to i. On the other hand, assume that (i, j) satisfies the right hand side, and let by contradiction $P \notin MCS(X_{<i}, Y_{<j})$. By definition of swings, $P \in MCS(X_{<i}, Y_{\leq m})$ and $P \in MCS(X_{\leq l}, Y_{<j})$, which means that we cannot re-map and perform insertions in these two pairs of strings. Thus, the only way for P to not be in $MCS(X_{<i}, Y_{<j})$ is inserting an edge (p, q) between edges (l, m) and (i, j), i.e., with $l < p < i$ and $m < q < j$, but this directly contradicts the fact that $(i, j) \in Ext_P$. □

Thus, if we can quickly update the swings of the growing prefixes (specifically, of their leftmost mappings), then we only only require constant additional time to perform our maximality check. In practice, the swing of a leftmost mapping can be computed inductively in the following fashion, leading to procedure COMPUTESWINGS (Algorithm 3):

- (Base case) Let $L = (i, j)$ be a single-edged mapping corresponding to some character d; we want to find $\ltimes((i, j))$. For each character c in $Y_{<j}$, we compute $r_c = first_{X_{>i}}(c)$; that is, r_c is the next occurrence of c after i in X: this corresponds to a possible insertion of c before d if we move i after r_c. Consider thus $r = \min_{c\in\Sigma}\{r_c\}$. At this point, the first occurrence of d in X after r (i.e. $first_{X_{>r}}(d)$) is the top swing for the mapping. Computation of the bottom swing is symmetrical.
- (Inductive case) Given a leftmost mapping $L = l_1, \ldots, l_N$ with its swings $\ltimes(L)$, we want to compute the swings for a leftmost mapping of the kind $L \cup (i, j) = l_1, \ldots, l_N, (i, j)$. We first need to compute the *personal swing* of the new edge (i, j) with respect to L, and compare it with the *cumulative swing* $\ltimes(L)$ of the previous mapping. Let $l_N = (l, m)$; the personal swing of edge (i, j) with respect to L, denoted with $\ltimes^L((i, j))$, is the swing of mapping $\{(i, j)\}$ performed over the strings $X_{>l}, Y_{>m}$, instead of over the whole strings X, Y. This swing tells us the change in the endpoints necessary to insert something between l_N and (i, j), and is analogous to the base case. The cumulative swing is simply the swing $\ltimes(L)$. At this point, the swing of the new mapping is given by the minimums of the two swings:

$$\ltimes(L \cup (i, j)) = (\min\{\ltimes_T(L), \ltimes_T^L((i, j))\}, \min\{\ltimes_B(L), \ltimes_B^L((i, j))\}).$$

Algorithm 3 (Updating the Swings of a Mapping)

```
Input: a leftmost mapping $L$, its swings $\ltimes(L)$ and an edge $(i, j)$ in leftmost form
Output: the swings $(\ltimes_T, \ltimes_B)$ of mapping $L \cup (i, j)$
1: procedure ComputeSwings($L$, $\ltimes(L)$, $(i, j)$)
2:     let $(l, m)$ be the last edge of mapping $L$
3:     for $c \in \Sigma$ occurring in both $Y_{>m}$ and $Y_{<j}$ do
4:         $r_c \leftarrow first_{X_{>i}}(c)$
5:     end for
6:     for $c \in \Sigma$ occurring in both $X_{>l}$ and $X_{<i}$ do
7:         $s_c \leftarrow first_{Y_{>j}}(c)$
8:     end for
9:     $r \leftarrow \min_{c\in\Sigma} t_c$
10:    $s \leftarrow \min_{c\in\Sigma} b_c$
11:    Let $(i, j)$ correspond to character $d$
12:    $t \leftarrow first_{X_{>r}}(d)$
13:    $b \leftarrow first_{Y_{>s}}(d)$
14:    output $(\min\{\ltimes_T(L), t\}, \min\{\ltimes_B(L), b\})$
15: end procedure
```

Procedure ComputeSwings performs the described operations, updating the swings of a mapping when an edge is added. We will adopt the convention that all leftmost mappings start with edge $(-1, -1)$, which by itself has swing $(|X|, |Y|)$. With this convention, the base case occurs when $L = \{(-1, -1)\}$: in this case we have $(l, m) = (-1, -1)$, which ensures that r_c, s_c are computed over the whole strings $X_{>-1} = X, Y_{>-1} = Y$ (lines 3, 6), and $\ltimes(L) = (|X|, |Y|)$, in turn ensuring that at line 14 the personal swing of (i, j) is returned.

Complexity analysis Every step of the procedure requires either $O(\sigma)$ (when we iterate over the alphabet), or constant time (looking for the first occurrences of a character in a string). Thus, the total complexity for updating the swings is of $O(\sigma)$.

5.3 Refined Algorithm for MCS Enumeration

We are now ready to present our refined enumeration algorithm (Algorithm 4). The algorithm implements the two improvements described in Sects. 5.1–5.2, employing the two procedures FindExtensions, ComputeSwings. Furthermore, one more refinement is given by lines 8–13, where the set of characters which correspond to at least one edge of Ext_P is computed. The recursive calls will be performed looping over this set, instead of the Ext set, ensuring in a more efficient way that no two recursive calls are performed with respect to the same character.

Just like its unrefined counterpart, Algorithm 4 employs a binary partition scheme over the characters that extend the current prefix in a valid way. The preprocessing phase is identical to the original one, as both the data structures and the FindUnshiftables procedure are unchanged. The recursive computation is performed by the procedure RefinedBinaryPartition, this time also keeping track of the swing of the mapping L_P of the current prefix P. Once again, we add the character $\# \notin \Sigma$ at positions $(-1, -1)$; the first recursive call is then performed with $P = \#$, and leftmost mapping $L_P = \{(-1, -1)\}$, with swing given by the last posi-

Algorithm 4 (Refined MCS enumeration)

```
    Input: two strings $X, Y$ over given alphabet $\Sigma$
    Output: $MCS(X, Y)$
 1: procedure RefinedMCS($X, Y, \Sigma$)
 2:     $\mathcal{U} = $ FindUnshiftables($(|X|, |Y|)$)
 3:     RefinedBinaryPartition(#, $\{(-1, -1)\}$, $(|X|, |Y|)$)
 4: end procedure

    Input: a valid prefix $P$, its leftmost mapping $L_P$, and its swing $\ltimes(L_P)$
    Output: all strings in $MCS(X, Y)$ having $P$ as prefix
 5: procedure RefinedBinaryPartition($P, L_P, \ltimes(L_P)$)
 6:     $(l, m) \leftarrow$ last edge of $L_P$
 7:     $\mathcal{E} = $ FindExtensions($(l, m)$)
 8:     $\mathcal{V} = \emptyset$
 9:     for $(i, j) \in \mathcal{E}$ do
10:         if $i \leq \ltimes_T(L_P)$ and $j \leq \ltimes_B(L_P)$ then
11:             add the character corresponding to $(i, j)$ to $\mathcal{V}$
12:         end if
13:     end for
14:     if $\mathcal{V} = \emptyset$ then return $P$
15:     else
16:         for $c \in \mathcal{V}$ do
17:             $i \leftarrow first_{X_{>l}}(c)$
18:             $j \leftarrow first_{Y_{>m}}(c)$
19:             $\ltimes = $ ComputeSwings($L_P, \ltimes(L_P), (i, j)$)
20:             RefinedBinaryPartition($P\,c, L_P \cup (i, j), \ltimes$)
21:         end for
22:     end if
23: end procedure
```

tions of the strings $(|X|, |Y|)$. This is considered the starting swing because trivially $P = \# \in MCS(X, Y_{\leq -1}), MCS(X_{\leq -1}, Y)$.

At each step, the procedure finds the set of valid extensions $\mathcal{E}$ by calling procedure FindExtensions (Algorithm 2). Then, at lines 8–13 it directly extracts the set $\mathcal{V}$ of valid characters that satisfy both conditions of Theorem 3. To this end, it loops over all edges of $\mathcal{E}$ checking for maximality through the swing property of Eq. (1). If the maximality condition is satisfied, the corresponding character is added to $\mathcal{V}$. If $\mathcal{V}$ is empty, then P has no valid extensions, and it is returned as it must be an MCS; otherwise, every character it contains produces a valid extension of P, and will thus produce a recursive call: for every character in $\mathcal{V}$, the procedure computes its first pairwise occurrences (i, j) after the mapping, updates the swings of $L_P \cup (i, j)$ by calling procedure ComputeSwings (Algorithm 3), and performs the recursive call corresponding to prefix $P\,c$ by passing on the new mapping $L \cup (i, j)$ and its updated swing. Being a refinement of Algorithm 1, the correctness of RefinedMCS is immediate.

Complexity analysis. The procedure starts by computing the extension set $\mathcal{E}$, by calling the $O(\sigma \log n)$-time procedure FindExtensions. Then, it computes set $\mathcal{V}$ by going through each of the $O(\sigma)$ elements of $\mathcal{E}$, and performing the constant-time maximality check of Eq. (1) (line 19). Thus, computing $\mathcal{V}$ requires $O(\sigma)$ time overall. Afterwards, for every element of $\mathcal{V}$, its first pairwise occurrences in $X_{>l}$, $Y_{>m}$ are computed. Once again, this can be performed in $O(1)$ time, using the string data structure. Lastly, before recurring, the $O(\sigma)$-time procedure ComputeSwings is called to update the swings of the new mapping. Thus, the time required to perform one recursive call of RefinedBinaryPartition is $O(\sigma \log n + \sigma^2)$. However, we can further improve this with the following observation: every time line 19 is executed, a new recursive call is immediately generated. We can thus simply attribute the cost of ComputeSwings to this call (as if the computation was performed in it), meaning that each recursive call will only have to pay the cost of a single ComputeSwings procedure, i.e., $O(\sigma)$ time instead of $O(\sigma^2)$ time, for a total time cost of

$$O(\sigma \log n + \sigma) = O(\sigma \log n).$$

As before, the delay of the final algorithm RefinedMCS is given by the cost of one recursive call times the height of the partition tree, which in our case is $O(n)$. Thus, algorithm RefinedMCS has a delay of

$$O(\sigma n \log n)$$

and requires $O(\sigma n^2 \log n)$ time for preprocessing. Regarding space complexity, in addition to the aforementioned data structures, we need to store other data for each recursive call in a root-to-leaf path in the partition tree. Namely, we store the sets $\mathcal{E}$ and $\mathcal{V}$, the last edge of the current leftmost mapping, and the values of the swings, leading to an additional $O(n\sigma)$ total space. The total space employed by the algorithm is therefore $O(n\sigma + |\mathcal{U}| + n) = O(n^2)$, and Theorem 1 is proved.

Specifically, with a constant-sized alphabet, our algorithm enumerates all MCS with delay $O(n \log n)$, preprocessing time $O(n^2 \log n)$, and quadratic space complexity.

6 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, JST CREST, Grant Number JPMJCR1401, and the Italian Ministry of University and Research (MIUR) under PRIN Project No. 20174LF3T8 AHeAD. 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. A preliminary version of the work in this paper appeared in [6].

We would like to thank Professor Shin-Ichi Minato (Kyoto University) for the useful discussions on combinatorial properties of MCS held during the WEPA 2018 workshop in Pisa, including those illustrated in Example 2.

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 (2015)
2. Abboud, A., Williams, V.V.: Popular conjectures imply strong lower bounds for dynamic problems. In: *2014 IEEE 55th Annual Symposium on Foundations of Computer Science*, pp. 434–443. IEEE (2014)
3. 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)
4. Bergroth, L., Hakonen, H., Raita, T.: A survey of longest common subsequence algorithms. In: *Proceedings 7th International Symposium on String Processing and Information Retrieval*, pp. 39–48. SPIRE (2000)
5. Chain, P., Kurtz, S., Ohlebusch, E., Slezak, T.: An applications-focused review of comparative genomics tools: capabilities, limitations and future challenges. Brief. Bioinf. **4**(2), 105–123 (2003)
6. Conte, A., Grossi, R., Punzi, G., Uno, T.: Polynomial-delay enumeration of maximal common subsequences. In: Brisaboa, N.R., Puglisi, S.J. (eds.) String Processing and Information Retrieval, pp. 189–202. Springer, Cham (2019)
7. Crochemore, M., Melichar, B., Tronıček, Z.: Directed acyclic subsequence graph-overview. J. Discrete Algorithms **1**(3–4), 255–280 (2003)
8. Delcher, A.L., Kasif, S., Fleischmann, R.D., Peterson, J., White, O., Salzberg, S.L.: Alignment of whole genomes. Nucl. Acids Res. **27**(11), 2369–2376 (1999)
9. Fraser, C.B., Irving, R.W., Middendorf, M.: Maximal common subsequences and minimal common supersequences. Inf. Comput. **124**(2), 145–153 (1996)
10. Hirschberg, D.S.: Algorithms for the longest common subsequence problem. J. ACM **24**(4), 664–675 (1977)
11. Hsu, W.J., Du, M.W.: Computing a longest common subsequence for a set of strings. BIT Numer. Math. **24**(1), 45–59 (1984)
12. Hunt, J.W., Szymanski, T.G.: A fast algorithm for computing longest common subsequences. Commun. ACM **20**(5), 350–353 (1977)
13. Impagliazzo, R., Paturi, R.: On the complexity of k-sat. J. Comput. Syst. Sci. **62**(2), 367–375 (2001)
14. 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)
15. Knuth, D.E.: The Art of Computer Programming, Volume 3: Sorting and Searching of Addison-Wesley series in computer science and information processing. Addison-Wesley (1997)
16. Kurtz, S., Phillippy, A., Delcher, A.L., Smoot, M., Shumway, M., Antonescu, C., Salzberg, S.L.: Versatile and open software for comparing large genomes. Genome Biol. **5**(2), 1–9 (2004)
17. Lawler, E.L., Lenstra, J.K., Rinnooy Kan, A.H.G.: Generating all maximal independent sets: NP-hardness and polynomial-time algorithms. SIAM J. Comput. **9**(3), 558–565 (1980)
18. Masek, W.J., Paterson, M.S.: A faster algorithm computing string edit distances. J. Comput. Syst. Sc. **20**(1), 18–31 (1980)
19. Raman, R., Raman, V., Satti, S.R.: Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Trans. Algorithms **3**(4), 43-es (2007)
20. Sakai, Y.: Maximal common subsequence algorithms. In: Navarro, G., Sankoff, D., Zhu, B. (eds.) Annual Symposium on Combinatorial Pattern Matching (CPM 2018), Volume 105 of Leibniz International Proceedings in Informatics (LIPIcs), Dagstuhl, Germany, pp. 1:1–1:10. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2018)
21. Sakai, Y.: Maximal common subsequence algorithms. Theor. Comput. Sci. **793**, 132–139 (2019)
22. Wagner, R.A., Fischer, M.J.: The string-to-string correction problem. J. ACM **21**(1), 168–173 (1974)