Finding Frequent Subsequences in a Set of Texts

Alban Mancheron, Jean-Émile Symphor

Finding Frequent Subsequences in a Set of Texts. [version 1.8.2.9]

Alban Mancheron[†] and Jean-Émile Symphor[‡]

[‡] inria, Centre Lille – Nord-Europe, France.
alban.mancheron@inria.fr
[‡] Grimaag, Université des Antilles et de la Guyane, Martinique, France.
je.symphor@martinique.univ-ag.fr

Abstract. Given a set of strings, the Common Subsequence Automaton accepts all common subsequences of these strings. Such an automaton can be deduced from other automata like the Directed Acyclic Subsequence Graph or the Subsequence Automaton. In this paper, we introduce some new issues in text algorithm on the basis of Common Subsequences related problems. Firstly, we make an overview of different existing automata, focusing on their similarities and differences. Secondly, we present a new automaton, the Constrained Subsequence Automaton, which extends the Common Subsequence Automaton, by adding an integer q denoted quorum.

1 Introduction

In many research areas (such as network monitoring, molecular biology, data mining), extracting all common subsequences from a set of texts is one of the major issues on today. It allows to characterize some properties of the set of texts. In the field of text algorithmic, a well known problem is to find the longest common subsequence of a set of strings. This problem is known as the lcs Problem. Before giving this problem statement and some connected one, we need to introduce some usual definitions. A sequence s of symbols taken in a set Σ is called a string; the set Σ is then called the alphabet. A subsequence of a string s is a string t such that t can be obtained from s by deleting zero or more symbols. Given a set of strings $\mathcal{S}$, the string t is called a common subsequence if t is a subsequence of every s from $\mathcal{S}$.

In 1986, Hébrard and Crochemore proposed an algorithm for building a deterministic and acyclic automaton that accepts all subsequences of a given string s [1]. Their algorithm processes the string s from right to left. They called this structure the Directed Acyclic Subsequence Graph (dasg). This way, Baeza-Yates extended the structure in 1991 for the case of multiple texts [2]. The dasg is now an acyclic and deterministic automaton that accepts all subsequences of any of the input texts. In the following, we denote the dasg of a set $\mathcal{S}$ by dasg($\mathcal{S}$). So, given a set of sequences $\mathcal{S}$, he presented a right-to-left algorithm for building dasg($\mathcal{S}$). Troníček and Melichar introduced

a left-to-right algorithm for constructing a DASG for one text and a quasi left-to-right algorithm for building DASG($\mathcal{S}$) in 1998 [3] (see also [4]). A left-to-right algorithm allows to construct such an automaton without parsing the whole texts from $\mathcal{S}$a priori. We use the quasi qualifier because, their algorithm processes the strings from left to right, but it requires to have a look straight forward in the sequences. In 2000, HOSHINO & al. [5] introduced a new structure close to the DASG, which accepts exactly the same language, which is still acyclic (with a slightly modification) and deterministic, called the Subsequence Automaton (SA); and they provide a left-to-right algorithm that constructs the structure. On the basis of these results, TRONÍČEK recently introduced an algorithm that builds an acyclic and deterministic automaton that accepts only the common subsequences of a set of texts from $\mathcal{S}$. He called this structure the Common Subsequence Automaton (CSA) [6]. In the following, we denote by CSA($\mathcal{S}$) the CSA build from $\mathcal{S}$. It is obvious that all these structures may help a lot for many of subsequence problems.

In this paper, we introduce some new issues in text algorithm on the basis of LCS related problems. The first one that we introduce can be stated as following: given a set of texts $\mathcal{S}$, and an integer q (denoted quorum) such that $1 \leq q \leq |\mathcal{S}|$, find the longest common subsequence of at least q string from $\mathcal{S}$. We denote this problem by LCS$_q$. This problem can be extended to other text algorithm issues, such as the shortest distinguishing subsequence problem (SDS). Recall this last: given two sets of texts $\mathcal{S}$ and $\mathcal{T}$, find the shortest common subsequence from $\mathcal{S}$ that is not a subsequence of any text from $\mathcal{T}$. Integrating a quorum constraint $1 \leq q \leq \mathcal{S}$ can be resumed as to find the shortest common subsequence of at least q strings from $\mathcal{S}$ that is not a subsequence of any text from $\mathcal{T}$ (we denote this problem SDS$_q$). This issue can also be extended with a second quorum constraint: given two sets of texts $\mathcal{S}$ (positive set) and $\mathcal{T}$ (negative set) and q_1, q_2 two integers such that $1 \leq q_1 \leq |\mathcal{S}|$ and $1 \leq q_2 \leq |\mathcal{T}|$, find the shortest common subsequence of at least q_1 strings from $\mathcal{S}$ that is not a subsequence of at least q_2 texts from $\mathcal{T}$ (we denote this problem by SDS$_{q_1,q_2}$ in the subsequent). All these issues have applications in many fields, such as molecular biology (e.g., the identification of haplotypes in chromosomes) or SPAM detection in mails.

The LCS problem can easily be solved using the CSA of the set of input strings. In the same way, finding a solution to the SDS problem can be achieved by building the CSA of the positive set of strings and the DASG (or SA) of the negative set of strings. It is obvious that solutions of the more general problems LCS$_q$, SDS$_q$ and SDS$_{q_1,q_2}$ can be computed using structures similar to CSA and DASG/SA. Indeed, the main result of this paper is the description of an automaton which is a generalization of both the CSA and the SA. Actually, we describe an acyclic and deterministic automaton which accepts all the subsequences common to at least q strings from a set $\mathcal{S}$. We call this structure a Constrained Subsequence Automaton with quorum q, and we denote it CSA$_q$. We obviously support our result of an algorithm.

The subsequent is organized as follow: second section presents the definitions of the DASG, the SA and the CSA; in the third section is formally introduced the CSA_q and we present an algorithm that build this structure; and the last one illustrates how the LCS_q and related problems can be solved using the CSA_q.

2 Requirements & Overview

Before giving a description of the existing automata mentioned in the above section, we first recall some notations and definitions. First recall that Σ denotes a finite alphabet, thus we denote by ε the empty string (of length 0). Given a string $w \in \Sigma^*$, we denote by $Sub(w)$ the set of all subsequences of w and by $|w|$ the length of w. We denote by $w[i]$ $(1 \leq i \leq |w|)$ the i^{th} character of w and by $w[i..j]$ $(1 \leq i \leq j \leq |w|)$ the substring of w starting at position i and ending at position j, that is $w[i..j] = w[i] \cdots w[j]$. A subsequence of a string w is any string obtained by deleting zero or more symbols from w.

We use in this paper the standard notation of finite automata [7]. A finite automaton is a 5-tuple $(\mathcal{Q}, \Sigma, \delta, \mathcal{I}, \mathcal{F})$ where $\mathcal{Q}$ is a finite set of states, Σ an input alphabet, $\delta : \mathcal{Q} \times \Sigma \to \mathcal{Q}$ is a transition function, $\mathcal{I} \subseteq \mathcal{Q}$ is the set of initial states and $\mathcal{F} \subseteq \mathcal{Q}$ is the set of final states.

2.1 The Directed Acyclic Subsequence Graph (original version)

Definition 1. *Given a string s of length n in Σ^*, an integer $p \in [0; n]$ and a symbol $\alpha \in \Sigma$ which at least once occurs in $s[p+1..n]$, we define the* D-*Reached Position (denoted $dRP_s(p, \alpha)$) as p', where $p' = \min\Big(\big\{j \mid j > p \wedge s[j] = \alpha\big\}\Big)$.*

This definition allows us to define the DASG for one string.

Definition 2. *Given a string s of length n in Σ^*, we define the Directed Acyclic Subsequence Graph for the string s as the 5-tuple $\text{DASG}(s) = (\mathcal{Q}, \Sigma, \delta, \mathcal{I}, \mathcal{F})$, where:*

$$\mathcal{Q} = [0; n], \qquad \mathcal{I} = \{0\}, \qquad \delta = dRP_s \qquad \text{and} \qquad \mathcal{F} = \mathcal{Q}$$

Thus, the $\text{DASG}(s) = (\mathcal{Q}, \Sigma, \delta, \mathcal{I}, \mathcal{F})$ accepts a string t if and only if t is a subsequence of s [3]. The automaton can be partial in the sense that each state needs not to have transitions for all $\alpha \in \Sigma$. We show an example of the $\text{DASG}(aba)$ in Fig. 1.

2.2 The Directed Acyclic Subsequence Graph (extended version)

Let $\mathcal{S}$ denote a set of texts $\{s_1, \ldots, s_k\}$. Let n_i be the length of s_i and $s_i[j]$ be j^{th} symbol of s_i for all $j \in [1; n_i]$ and all $i \in [1; k]$. We say that t is a subsequence of $\mathcal{S}$ if and only if some $i \in [1; k]$ exists such that t is a subsequence of s_i. The

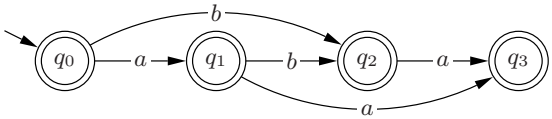

Fig. 1. DASG(aba) (original version).

DASG can be extended to a deterministic finite automaton which accepts all subsequences of s_i for all $i \in [1;k]$.

So, each state of the DASG corresponds to positions in texts. We start the reading by setting a "cursor" in front of the first symbol of each text. This set of cursor positions corresponds to the initial state of the automaton. For a given set of cursor positions, we check for all $\alpha \in \Sigma$ if α can be read from this cursor positions in at least one sequence. Thus, when a new symbol is processed, the position in texts may change. For a symbol α, the new position is obtained by searching the first α after the current position in each text. If there is no such symbol after the current position in some string s_i, we set the current reading position in this string after the last symbol of s_i (i.e., no more symbol can be read from this position in this string). The following definition formally state what is the set of available cursor positions.

Definition 3. *Given a set of strings $\mathcal{S} = \{s_1, \ldots, s_k\}$ from Σ^*, we define a position point of the set $\mathcal{S}$ as an ordered k-tuple $[p_1, \ldots, p_k]$, where $p_i \in [0, n_i]$ is a position in string s_i. If $p_i = 0$, then it denotes the empty string ε in front of the first position of s_i, otherwise it denotes the position of the $p_i{}^{th}$ symbol of s_i, for all $i \in [1;k]$.*

The particular position where $p_i = 0$ for all $i \in [1;k]$ is called the initial position point (denoted q_0^k in the subsequent). We also denote by $Pos(\mathcal{S})$ the set of all position points of $\mathcal{S}$.

Definition 4. *Given a set of strings $\mathcal{S} = \{s_1, \ldots, s_k\}$ from Σ^* and a position point $[p_1, \ldots, p_k] \in Pos(\mathcal{S})$, we define the subsequence position alphabet as the subset of Σ composed of all symbols which are contained in texts $s_i[p_i + 1..n_i]$ for all $i \in [1;k]$. We denote this set by:*

$$\Sigma_{\mathcal{S}}\big([p_1, \ldots, p_k]\big) = \bigcup_{i=1}^{k} \{\alpha \in \Sigma \mid \exists\, j \in [p_i + 1; n_i], s_i[j] = \alpha\}.$$

The "jump" between two position points of $Pos(\mathcal{S})$ when reading a symbol α can easily be stated from the two previous definitions.

Definition 5. *Given a set of strings $\mathcal{S} = \{s_1, \ldots, s_k\}$ from Σ^*, a position point $[p_1, \ldots, p_k] \in Pos(\mathcal{S})$ and $\alpha \in \Sigma_{\mathcal{S}}\big([p_1, \ldots p_k]\big)$, we define the D-Reached Position Point (denoted $dRPP_{\mathcal{S}}\big([p_1, \ldots, p_k], \alpha\big)$) as $[p'_1, p'_2, ..., p'_k]$, where $\forall\, i \in [1;k]$, $p'_i = \min\Big(\big\{j \mid j > p_i \wedge s_i[j] = \alpha\big\} \cup \big\{n_i\big\}\Big)$.*

Actually, we now can formally define the DASG of a set of texts.

Definition 6. *Given a set of strings $\mathcal{S} = \{s_1, \ldots, s_k\}$ from Σ^*, we define the Directed Acyclic Subsequence Graph for the strings $s_1, \ldots, s_k$ as the 5-tuple* $\text{DASG}(\mathcal{S}) = (\mathcal{Q}, \Sigma, \delta, \mathcal{I}, \mathcal{F})$*, where:*

$$\mathcal{Q} = Pos(\mathcal{S}), \qquad \mathcal{I} = \{q_0^k\}, \qquad \delta = dRPP_{\mathcal{S}} \qquad \textit{and} \qquad \mathcal{F} = Pos(\mathcal{S})$$

Naturally, the $\text{DASG}(\mathcal{S})$ accepts a string t if and only if $t \in \bigcup_{s \in \mathcal{S}} Sub(s)$. We illustrate in Fig. 2 an example of DASG for three texts. The string $s_1 = aba$ is the one that we use in Fig. 1. We choose for the second string $s_2 = aabb$ and $s_3 = aab$ for the third.

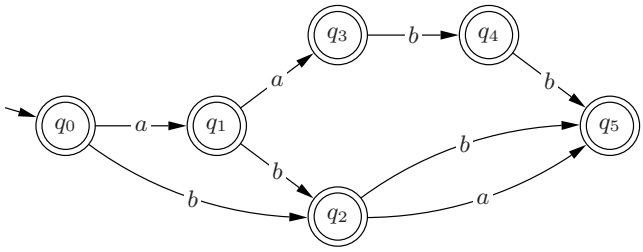

Fig. 2. $\text{DASG}(\{aba, aabb, aab\})$, where $q_0 := [0, 0, 0]$, $q_1 := [1, 1, 1]$, $q_2 := [2, 3, 3]$, $q_3 := [3, 2, 2]$, $q_4 := [3, 3, 3]$ and $q_5 := [3, 4, 3]$.

2.3 The Subsequence Automaton

HOSHINO & al. [5] introduced an algorithm for building an new deterministic complete finite automaton that recognizes exactly the same language than the DASG for a given set of strings. Since their structure is complete, they introduce a sink state, which is the only one that have cycles; so their automaton can be considered as acyclic, by not considering this state.

An important aspect of this automaton remains from the fact they introduce in their algorithm, a major difference in comparison with the DASG. Indeed, in the processing of position points in texts for a given symbol, they denote by ∞ the fact that the symbol of the alphabet doesn't occur in a string s_i after the position p_i, instead of using n_i (see definition 3).

More formally, for a position point (i.e., a k-tuple $[p_1, \ldots, p_k]$), where $p_i \in [0, n_i]$ a position in string s_i, the main difference compared to definition 3 appears in the case $p_i = n_i$. Indeed, it denotes the position of the last symbol of s_i if and only if this last symbol is the currently processed one. By opposite, they denote by ∞ the particular case for which the current processed symbol doesn't occur in $s_i[p_i + 1..n_i]$.

The definition of the position point can easily be modified by adding the ∞ value, which corresponds to the last position after the last symbol of any string s_i. Thus, that means that the symbol looked for in the alphabet doesn't exist in the rest of the string. So, it induces a new set of position points, which is a superset of $Pos(\mathcal{S})$: this new set is called $Pos'(\mathcal{S})$.

Definition 7. *Given a set of strings $\mathcal{S} = \{s_1, \ldots, s_k\}$ from Σ^*, we define an extended position point of the set $\mathcal{S}$ as an ordered k-tuple $[p_1, \ldots, p_k]$, where $p_i \in [0, n_i] \cup \{\infty\}$ is a position in string s_i. If $p_i = 0$, then it denotes the empty string ε in front of the first position of s_i, if $p_i = \infty$, then it denotes the empty string behind the last position of s_i, otherwise it denotes the position of the ${p_i}^{th}$ symbol of s_i, for all $i \in [1; k]$.*

This slight modification induces the creation of the particular state qualified as a "sink state", which corresponds to the extended position point where all $p_i = \infty$ (denoted q^k_∞). As a matter of fact, the use of the subsequence alphabet becomes obsolete. Consequently, the D-Reached Position Point has to be adapted too.

Definition 8. *Given a set of strings $\mathcal{S} = \{s_1, \ldots, s_k\}$ from Σ^*, an extended position point $[p_1, \ldots, p_k] \in Pos'(\mathcal{S})$ and $\alpha \in \Sigma$, we define the S-Reached Position Point (denoted $sRPP_{\mathcal{S}}([p_1, \ldots, p_k], \alpha)$) as $[p'_1, p'_2, ..., p'_k]$, where $\forall\, i \in [1; k]$, $p'_i = \min\Big(\big\{j \mid j > p_i \wedge s_i[j] = \alpha\big\} \cup \big\{\infty\big\}\Big)$.*

This idea is inducted in the TRONÍČEK approach [4], as he avoids the creation of the sink state by taking in consideration the subsequence position alphabet. Indeed, there is no transition created for the symbols non belonging to this alphabet, that is the case for the symbols presenting transitions leading to the sink state.

Definition 9. *Given a set of strings $\mathcal{S} = \{s_1, \ldots, s_k\}$ from Σ^*, we define the Subsequence Automaton for the strings $s_1, \ldots, s_k$ as the 5-tuple $\text{SA}(\mathcal{S}) = (\mathcal{Q}, \Sigma, \delta, \mathcal{I}, \mathcal{F})$, where:*

$$\mathcal{Q} = Pos'(\mathcal{S}), \quad \mathcal{I} = \{q^k_0\}, \quad \delta = sRPP_{\mathcal{S}} \quad \text{and} \quad \mathcal{F} = Pos'(\mathcal{S}) \setminus \{q^k_\infty\}$$

Such a SA accepts a string t if and only if $t \in \bigcup_{s \in \mathcal{S}} Sub(s)$. We illustrate in Fig. 3 an example of SA for the same strings than the previous example: $s_1 = aba$, $s_2 = aabb$ and $s_3 = aab$.

2.4 The Common Subsequence Automaton

Given a set of strings $\mathcal{S}$, a string t is a common subsequence of $\mathcal{S}$ if and only if t is a subsequence of every string from $\mathcal{S}$.

According to the definition of the set $Pos'(\mathcal{S})$ of extended position point, the value ∞ matches the situation which the currently processed symbol of the

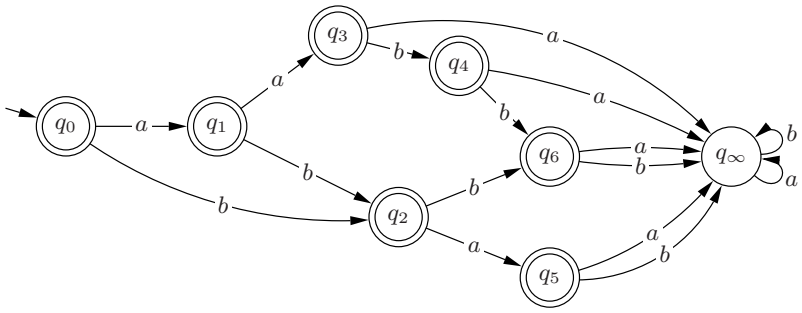

Fig. 3. SA($\{aba, aabb, aab\}$), where $q_0 := [0,0,0]$, $q_1 := [1,1,1]$, $q_2 := [2,3,3]$, $q_3 := [3,2,2]$, $q_4 := [\infty,3,3]$, $q_5 := [3,\infty,\infty]$, $q6 := [\infty,4,\infty]$ and $q_\infty := [\infty,\infty,\infty]$.

alphabet doesn't occur. So, given a state q in SA($\mathcal{S}$) and $\alpha \in \Sigma$, such that ∞ occurs in $sRPP_{\mathcal{S}}(q,\alpha)$, for any string t spelled out by a path from the initial state to the state q in SA($\mathcal{S}$), the string $t\,\alpha$ is not a subsequence of at least one string from $\mathcal{S}$. Thus, the CSA can be obviously deduced from the SA by pruning all states associated to a position point which contains the value ∞.

We can deduce the CSA from the DASG too by deriving the common subsequence position alphabet from definition 4.

Definition 10. *Given a set of strings $\mathcal{S} = \{s_1, \ldots, s_k\}$ from Σ^* and a position point $[p_1, \ldots, p_k] \in Pos(\mathcal{S})$, we define the common subsequence position alphabet as the subset of Σ composed of all symbols which are simultaneously contained in texts $s_i[p_i + 1..n_i]$ for all $i \in [1;k]$. We denote this set by:*

$$\Sigma'_{\mathcal{S}}\big([p_1, \ldots, p_k]\big) = \bigcap_{i=1}^{k} \{\alpha \in \Sigma \mid \exists\, j \in [p_i + 1; n_i], s_i[j] = \alpha\}.$$

We need to define what is a reachable position, as for DASG or SA.

Definition 11. *Given a set of strings $\mathcal{S} = \{s_1, \ldots, s_k\}$ from Σ^*, a position point $[p_1, \ldots, p_k] \in Pos(\mathcal{S})$ and $\alpha \in \Sigma'_{\mathcal{S}}\big([p_1, \ldots, p_k]\big)$, we define the C-Reached Position Point (denoted $cRPP_{\mathcal{S}}([p_1, \ldots, p_k], \alpha)$) as $[p'_1, p'_2, ..., p'_k]$, where $\forall\, i \in [1;k]$, $p'_i = \min\Big(\big\{j \mid j > p_i \wedge s_i[j] = \alpha\big\}\Big)$.*

From now, we can formally define the Common Subsequence Automaton.

Definition 12. *Given a set of strings $\mathcal{S} = \{s_1, \ldots, s_k\}$ from Σ^*, we define the Common Subsequence Automaton for the strings $s_1, \ldots, s_k$ as the 5-tuple CSA($\mathcal{S}$) $= (\mathcal{Q}, \Sigma, \delta, \mathcal{I}, \mathcal{F})$, where:*

$$\mathcal{Q} = Pos(\mathcal{S}), \qquad \mathcal{I} = \{q_0^k\}, \qquad \delta = cRPP_{\mathcal{S}} \qquad \text{and} \qquad \mathcal{F} = Pos(\mathcal{S})$$

Such a CSA accepts a string t if and only if $t \in \bigcap_{s \in \mathcal{S}} Sub(s)$. We illustrate in Fig. 4 an example of CSA for the same strings than the previous examples: $s_1 = aba$, $s_2 = aabb$ and $s_3 = aab$.

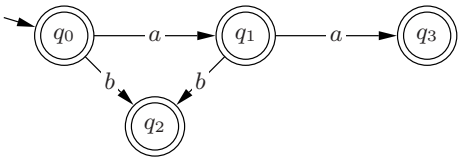

Fig. 4. CSA($\{aba, aabb, aab\}$), where $q_0 := [0, 0, 0]$, $q_1 := [1, 1, 1]$, $q_2 := [2, 3, 3]$ and $q_3 := [3, 2, 2]$.

3 The Constrained Subsequence Automaton

In this section, we present the definition of the CSA_q on the basis of the previously defined structure. We first introduce the quorum constraint notion and then, we show that a partial ordered relationship exists between the states of the CSA_q. We end this section by giving an algorithm for building this structure.

3.1 Defining the CSA_q for a Set of Strings

Informally, the quorum constraint q is satisfy for a string t if it is a subsequence of at least q texts of a set of texts $\mathcal{S}$. That means that given an automaton accepting all and only the subsequences of a set $\mathcal{S}$ satisfying a quorum constraint q, each path from the initial state to any state of the automaton spells out a subsequence of at least q strings from $\mathcal{S}$. Thus, according to the definition 7 there is at least q positions in the position point that are not ∞.

Definition 13. *Given a set of k strings $\mathcal{S}$ and an integer q such that $1 \leq q \leq k$, we says that a position point $[p_1, \ldots, p_k] \in Pos'(\mathcal{S})$ satisfies the quorum constraint q if and only if*

$$\Big|\{p_i \,|\, p_i < \infty\}\Big| \geq q,$$

and then, we say that $[p_1, \ldots, p_k]$ is a q-satisfying position point. We denote the subset of all states from $Pos'(\mathcal{S})$ that are q-satisfying by $Pos'_q(\mathcal{S})$.

In the following, since the states of the automata previously described are directly associated to the position points, we say that a state is q-satisfying if its corresponding position point is in $Pos'_q(\mathcal{S})$. Finally, by considering the SA of a set of strings $\mathcal{S}$, we now establish an important property from which we deduce the definition of the $\text{CSA}_q(\mathcal{S})$.

Proposition 1. *Given a set of k strings $\mathcal{S}$, its* SA *and an integer q such that $1 \leq q \leq k$, each path from the initial state of* SA($\mathcal{S}$) *to a q-satisfying state only goes through q-satisfying states. This is due to the fact that from a state which doesn't satisfy the constraint quorum q, there is no path that leads to a q-satisfying state.*

Proof. Let $[p_1, \ldots, p_k]$ a position point which doesn't satisfy the constraint quorum. That means that the text t spelled out by the path from the state q_0^k to the state $[p_1, \ldots, p_k]$ is not a subsequence of any strings from $\mathcal{S}$. Now, suppose that there is a path from $[p_1, \ldots, p_k]$ that leads to a q-satisfying state $[p'_1, \ldots, p'_k]$, which labeled the text t'. That would mean the text $t\,t'$ is a subsequence of a string from $\mathcal{S}$. Since if a text u is a subsequence of a string v, all the subsequences of u are also subsequences of v, that implies that each subsequence of $t\,t'$ is a subsequence of a string from $\mathcal{S}$. By definition, t is a subsequence of $t\,t'$, so the state $[p'_1, \ldots, p'k]$ couldn't exist. □

This property directly allows to conclude that if a path reaches a state which is not q-satisfying, then all paths from this state necessary leads to the sink state q_∞^k without going through a q-satisfying state.

Definition 14. *Given a set of strings $\mathcal{S} = \{s_1, \ldots, s_k\}$ from Σ^*, we define the Constrained Subsequence Automaton for the strings $s_1, \ldots, s_k$ and quorum q as the 5-tuple $\text{CSA}_q(\mathcal{S}) = (\mathcal{Q}, \Sigma, \delta, \mathcal{I}, \mathcal{F})$, where:*

$$\mathcal{Q} = Pos'(\mathcal{S}), \qquad \mathcal{I} = \{q_0^k\}, \qquad \delta = sRPP_{\mathcal{S}} \qquad \textit{and} \qquad \mathcal{F} = Pos'_q(\mathcal{S})$$

Finally, one can easily observe that for a set of k strings $\mathcal{S}$, setting the quorum constraint to $q = 1$ makes $\text{CSA}_1(\mathcal{S})$ being $\text{SA}(\mathcal{S})$ and setting the quorum constraint $q = k$ makes $\text{CSA}_k(\mathcal{S})$ being $\text{CSA}(\mathcal{S})$.

3.2 Building the CSA_q for a Set of Strings

Since we provide the definition of the CSA_q for a set of strings. We give in here a quasi left-to-right algorithm (see introduction) which, given a set of strings and a quorum constraint q, build the corresponding CSA_q.

First of all, since the CSA_q only differs from the SA with the set of the final states, one can think about using the SA construction algorithm [5] and only modify the set of final states during the construction. We quickly discuss this method in the conclusion. It is obvious that property 1 allows to consider only the final states of the CSA_q. Thus we only have to build them. We base our approach on the DASG (extended version) construction[1]. We first need to describe the BUILD_EXTENDED_POSITION_POINT method in order to return an extended position point from $Pos'_q(\mathcal{S})$ and not only from $Pos(\mathcal{S})$ (see algorithm 1.1). If the algorithm returns the special point q_0^k, that means the point given by function $sRPP$ is not q-satisfying.

Well, we slightly modify the BUILD_DASG algorithm in the following way:

- we only create the reachable states which are q-satisfying;
- we process each state in order to re-label them (more efficiency than setting the position point as label) as soon as possible.

[1] We do not provide the original algorithms, but we use dark red color – or grey inB&W mode – to illustrates the differences from these. They are given in [4].

Algorithm 1.1. BUILD_EXTENDED_POSITION_POINT

```
1  Inputs: S = {s_1, ..., s_k} % Set of strings. %
2    q ∈ [1;k] % Quorum constraint. %
3    pp = [p_1, ..., p_k] ∈ Pos'(S) % Extended position point. %
4    α ∈ Σ % Current processed symbol. %
5  Output: pp' = [p'_1, ..., p'_k] ∈ Pos'_q(S) % Extended position point. %
6  Variables: i % Sequence number. %
7    found % Boolean. %
8    cpt % Constraint satisfaction counter. %
9  Begin
10   cpt ← 0
11   For i ← 1 To k Do
12     p'_i ← min({j | j > p_i ∧ s_i[j] = α} ∪ {∞})
13     If p'_i ≠ ∞ Then
14       cpt ← cpt + 1
15     End If
16   End For
17   If cpt < quorum Then
18     pp' ← q_0^k
19   End If
20 End
```

The first item requires only to count the number of non-∞ values in potential position points and to directly remove it if it is not q-satisfying. The second item is subtler and we use a partial order relationship between the states of the CSA_q to integrate it. The following property establishes this partial order relationship. It is an extension of the relationship induced by the transition function δ and is true for either the DASG or the SA or the CSA.

Proposition 2. *Given a set of k strings $\mathcal{S}$, there is a partial order relationship between the extended position points from $Pos'(\mathcal{S})$ induced by the transition function $sRPP_{\mathcal{S}}$. Let two extended position points $pp = [p_1, \ldots, p_k]$ and $pp' = [p'_1, \ldots, p'k]$. We say that the state pp is lower or equal to the state pp' if and only if for all $[i \in [1;k]$, $p_i < p'_i$. We denote this relation by $pp \preccurlyeq pp'$. Moreover, if $pp \neq pp'$ (i.e., $\exists\, i \in [1;k]$ such that $p_i < p'_i$) , we say that the state pp is lower than pp' and we denote this relationship by $pp \prec pp'$. Finally, given a set of extended position points $PP \in Pos'(\mathcal{S})$, we denote by $\min(PP)$ the subset of PP such that:*

$$\min(PP) = \{pp \mid \nexists pp' \in PP, pp' \preccurlyeq pp\}.$$

We say that $\min(PP)$ is the set of all the minimal states of PP.

It is obvious that if we consider two states q and q' such that there exists a path from q that reaches q', we necessary observe that $q \preccurlyeq q'$, and if q is not $[p_\infty, \ldots, p_\infty]$, $q \prec q'$.

Algorithm 1.2. BUILD_CSA$_q$

```
1  Inputs: S % Set of k strings {s_1,...,s_k}. %
2    q ∈ [1;k] % Quorum constraint. %
3  Output: CSA_q(S) % Acyclic and deterministic automaton %
4                   % accepting exactly all subsequences of at %
5                   % least q strings from S. %
6  Variables: α ∈ Σ % Symbol of the alphabet. %
7    Queue % Set of extended position points to process. %
8    pp, pp' % Extended position points. %
9    id % State id (for space optimization). %
10 Begin
11   Queue ← {q_0^k}
12   id ← 0
13   While Queue ≠ ∅ Do
14     Let pp be a minimal extended position point from Queue.
15     Add state pp, rename it as id and mark it as final.
16     For Each α ∈ Σ Do
17       pp' ← BUILD_EXTENDED_POSITION_POINT(S, q, pp, α)
18       If pp' ≠ q_0^k Then
19         Queue ← Queue ∪ {pp'}
20         Add a transition labeled by α from id to pp'.
21       End If
22     End For
23     id ← id + 1
24     Queue ← Queue \ {pp}
25   End While
26 End
```

We illustrate in Fig. 5 an complete example of CSA$_q$ construction for the same strings than the previous examples: $s_1 = aba$, $s_2 = aabb$, $s_3 = aab$ and with a quorum constraint $q = 2$.

If we consider the space complexity results provided in [8], since the CSA$_q$ has at least as many states than the CSA, it is obvious that the number of states of the CSA$_q$ is $T = \Omega\left(\frac{n^k}{(k+1)^k\, k!}\right)$. The time complexity is trivially the same as the DASG construction algorithm: $O\left(T\ |\Sigma|\ k\, c\right)$ where c is the cost charged for finding the position p'_i from $s_i[p_i..n_i]$. This cost is $O(1)$ by using a $|\Sigma| \times n$ matrix for each sequence (filled in a preprocessing step).

4 Some Applications

Recall that the LCS$_q$ problem can be state as following: given a set of texts $\mathcal{S} = \{s_1, \ldots, s_k\}$ and an integer q such that $1 \leq q \leq k$, the problem is to find the longest common subsequence of at least q strings from $\mathcal{S}$. This issue can

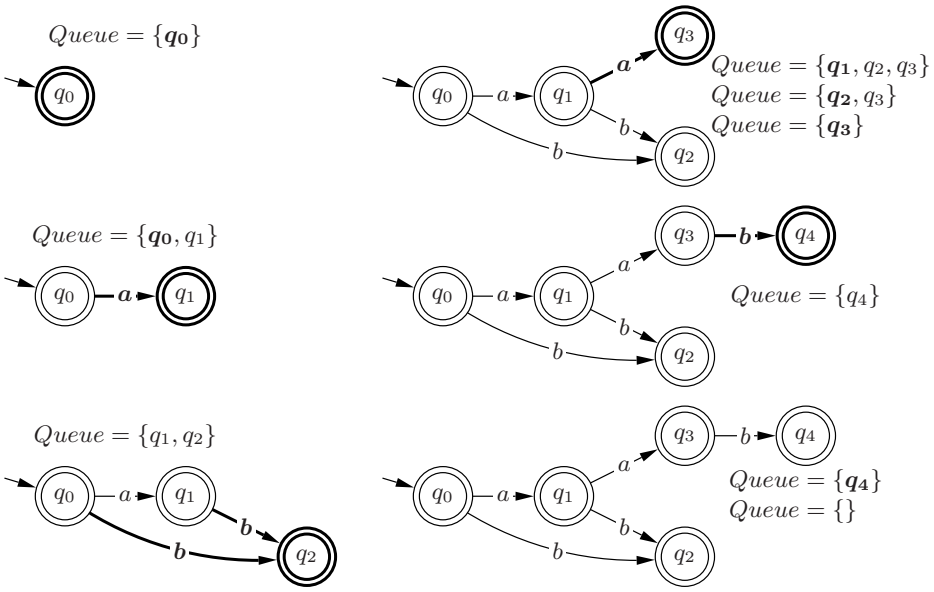

Fig. 5. $\text{CSA}_2(\{aba, aabb, aab\})$, where $q_0 := [0, 0, 0]$, $q_1 := [1, 1, 1]$, $q_2 := [2, 3, 3]$, $q_3 := [3, 2, 2]$ and $q_4 := [\infty, 3, 3]$.

easily be solved by constructing the $\text{CSA}_q(\mathcal{S})$ and by looking at the longest path starting from the initial state that leads to a final state.

The SDS_q problem can be state as following: given two sets of texts $\mathcal{S} = \{s_1, \ldots, s_k\}$ and $\mathcal{T} = \{t_1, \ldots, t_\ell\}$ and an integer q such that $1 \leq q \leq k$, the problem is to find the shortest common subsequence of at least q strings from $\mathcal{S}$, which is not a subsequence of any text from $\mathcal{T}$. This problem can be achieved using on the first hand the $\text{CSA}_q(\mathcal{S})$ and on the second hand the $\text{DASG}(\mathcal{T})$. Actually, the shortest distinguishing subsequence is the shortest sequence accepted by $\text{CSA}_q(\mathcal{S})$, which is rejected by $\text{DASG}(\mathcal{T})$.

This last issue is a direct extension of the previous one. It can be state as following: given two sets of texts $\mathcal{S} = \{s_1, \ldots, s_k\}$ and $\mathcal{T} = \{t_1, \ldots, t_\ell\}$ and two integers q_1 and q_2 such that $1 \leq q_1 \leq k$ and $1 \leq q_2 \leq \ell$, the problem is to find the shortest common subsequence of at least q strings from $\mathcal{S}$, which is not a subsequence of at least q_2 texts from $\mathcal{T}$. The SDS_q problem is a special case of the SDS_{q_1,q_2}, where $q_1 = q$ and $q_2 = \ell$. For solve this problem, we need to build the $\text{CSA}_{q_1}(\mathcal{S})$ and the $\text{CSA}_{q_2}(\mathcal{T})$. Thus, a sequence w is a solution of SDS_{q_1,q_2} if it is the shortest sequence accepted by $\text{CSA}_{q_1}(\mathcal{S})$ and rejected by $\text{CSA}_{q_2}(\mathcal{T})$. Effectively, in the special case where $q_2 = 1$, the $\text{CSA}_1(\mathcal{T})$ is the $\text{SA}(\mathcal{T})$, which accepts exactly the same language than the $\text{DASG}(\mathcal{T})$.

5 Conclusion

Our first motivation was to carry out an overview of the various structures that accepts subsequences of a set of strings and to illustrate their similarities and their differences. Our second motivation, by introducing the CSA_q, is justified by our need to look for subsequences, which occurs or doesn't in a subset of input texts. HOSHINO & al. [5] provide a left-to-right algorithm for the $\text{SA}(\mathcal{S})$ construction. This algorithm could be used in order to build the $\text{CSA}_q(\mathcal{S})$, but since the sequences from $\mathcal{S}$ are processed one by one, pruning the non q-satisfying states can't be operated before $k-q+1$ strings have been done. Actually, a left-to-right solution based on the SA construction can be considered to build an approximated CSA_q on the basis of a heuristic pruning strategy. Unfortunately, efficient heuristics are closely correlated to the distribution lows of the symbols in the texts. We aim to investigate the formal properties of the CSA_q in order to develop an approximated structure, as (for example) a probabilistic automaton.

References

1. HÉBRARD, J.J., CROCHEMORE, M.: Calcul de la distance par les sous-mots. Informatique Théorique et Applications **20**(4) (1986) 441–456
2. BAEZA-YATES, R.: Searching subsequences. Theoretical Computer Science (TCS) **78**(2) (1991) 363–376
3. TRONÍČEK, Z., MELICHAR, B.: Directed Acyclic Subsequence Graph. In HOLUB, J., ŠIMÁNEK, M., eds.: Proceedings of the Prague Stringology Club Workshop '98, Czech Technical University in Prague, Czech Republic (1998) 107–118
4. CROCHEMORE, M., MELICHAR, B., TRONÍČEK, Z. : Directed Acyclic Subsequence Graph – Overview. Journal of Discrete Algorithms **1**(3–4) (2003) 255–280
5. HOSHINO, H., SHINOHARA, A., TAKEDA, M., ARIKAWA, S.: Online Construction of Subsequence Automata for Multiple Texts. In: Proceedings of the 7th International Symposium on String Processing and Information Retrieval. (2000) 146–152
6. TRONÍČEK, Z.: Common Subsequence Automaton. In CHAMPARNAUD, J.M., MAUREL, D., eds.: Proceedings of the 7th International Conference on Implementation and Application of Automata. Number 2608 in Lecture Notes in Computer Science (LNCS), Tours, France, SPRINGER-VERLAG (2002) 270–275
7. HOPCROFT, J.E., ULLMAN, J.D.: Introduction to Automata Theory, Languages, and Computation. ADDISON-WESLEY Longman Publishing Co., Inc., Boston, USA (1990)
8. TRONÍČEK, Z., SHINOHARA, A.: The Size of Subsequence Automaton. Theoretical Computer Science (TCS) **341**(1–3) (2005) 379–384