View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Elsevier - Publisher Connector

Journal of Discrete Algorithms 8 (2010) 117–130

Contents lists available at ScienceDirect

Journal of Discrete Algorithms

www.elsevier.com/locate/jda

Finite automata based algorithms on subsequences and supersequences of degenerate strings

Costas Iliopoulos [a,*], M. Sohel Rahman [a], Michal Voráček [b], Ladislav Vagner [b]

[a] Algorithm Design Group, Department of Computer Science, King's College London, Strand, London WC2R 2LS, United Kingdom
[b] Department of Computer Science & Engineering, Czech Technical University, Czech Republic

ARTICLE INFO

Article history:
Received 8 January 2008
Accepted 1 October 2008
Available online 22 September 2009

Keywords:
Finite automata
Degenerate strings
Longest common subsequence
Shortest common supersequence
Constrained longest common subsequence

ABSTRACT

In this paper, we present linear-time algorithms for the construction two novel types of finite automata and show how they can be used to efficiently solve the *Longest Common Subsequence* (LCS), *Shortest Common Supersequence* (SCS) and *Constrained Longest Common Subsequence* (CLCS) problems for *degenerate strings*.

© 2009 Elsevier B.V. All rights reserved.

1. Introduction

In this paper, we present efficient algorithms to construct two novel types of finite automata on degenerate strings, namely the *subsequence automaton (SubAtm)* and the *supersequence automaton (SuperAtm)* and show how they can be used to efficiently solve three classic problems in computer science. In particular, we consider the classic and well-studied Longest Common Subsequence (LCS) and the Shortest Common Supersequence (SCS) problems along with a recent interesting variant of the former, namely the Constrained LCS (CLCS) problem. Note however that, all the problems considered here are in a different setting, i.e., for degenerate strings. We first present algorithms for the construction of the above-mentioned two automata, both of which work in linear time with respect to the length of the given degenerate strings. Notably, the subsequence automaton for degenerate strings can be seen as an extension of subsequence automaton for (normal) strings also known as *direct acyclic subsequence graph (DASG)* in the literature. A finite automaton accepting all subsequences of a given string was first mentioned by Hébrard and Crochemore [9]. Baeza-Yates proposed its right to left construction [3] and Troníček and Melichar presented its left to right construction [24]. Using the efficient construction of SubAtm and SuperAtm, we then present efficient algorithms for the LCS, SCS and CLCS problems for degenerate strings.

LCS and SCS problems are two heavily studied classic problems in computer science, both having extensive applications in diverse areas. Given a string, a subsequence is formed by deleting some of the characters of the string without disturbing the relative positions of the remaining characters. In such a case, the given string is said to be the supersequence of the formed subsequence. Given a set of strings, the LCS problem aims to compute a subsequence of maximum length, which is common to all the strings of the set. On the other hand, the goal of the SCS problem is to compute a string of minimum length having each of the strings of the set as its subsequence. The LCS length provides a convenient measure of similarity for long molecules, considered as nucleotide sequences [21], and also for arbitrary sequential objects, such

* Corresponding author.
E-mail addresses: csi@dcs.kcl.ac.uk (C. Iliopoulos), sohel@dcs.kcl.ac.uk (M.S. Rahman), voracem@fel.cvut.cz (M. Voráček), xvagner@fel.cvut.cz (L. Vagner).

1570-8667/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.jda.2008.10.004

as files. Another application of LCS can be found in text editing dealing with the problem of converting one word into another by a minimum number of edit operations – deletion and insertion of a letter, replacement of a letter with another, transposition of neighboring letters [26,16]. Data compression [20] requires storing a large number of similar files with maximum space economy. This can be achieved by using the file LCS or SCS with a collection of shorter codes for restoring the files. In cluster analysis, LCS can be used to estimate the closeness of an object to a template [17]. SCS and LCS problems find applications in mechanical engineering [15] as well. Here, SCSs play a special role: it is used to identify the shortest standardized technological process for a given set of separate technological processes, viewed as sequences of processing operations [22,4]. Apart from above classic references, applications of LCS and SCS, especially in bioinformatics, can be found in [8] and references therein.

The LCS problem for k strings ($k \geqslant 2$) (k-LCS) was first shown to be NP-hard [18] and later proved to be hard to be approximated [14]. More specifically, Jiang and Li, in [14], showed that there exists a constant $\delta > 0$ such that if k-LCS problem has a polynomial time approximate algorithm with performance ratio n^{δ}, then $P = NP$. The restricted but probably the more studied problem that deals with two strings (2-LCS) has been studied extensively (see [5] for a survey). The classic dynamic programming solution to 2-LCS problem, invented by Wagner and Fischer [26], has $O(n^2)$ worst case running time, assuming, for the sake of simplicity, that the given strings are of equal length, n. The dynamic programming approach was generalized by Itoga [13] to get a solution for k-LCS problem in $O(kn^k)$ time. Like the k-LCS problem, k-SCS problem is also NP-hard and hard to be approximated [14]. Interestingly, the 2-SCS problem can be easily solved from the solution of corresponding 2-LCS problem: in the solution of LCS, we just need to insert the 'non-lcs' characters preserving the order of the characters. Also, Timkovsky [23] showed that SCS problem can be solved using the dynamic programming approach in the same running time as LCS problem.

CLCS problem, on the other hand, is a relatively newer variant of the classic LCS problem. In CLCS, the computed longest common subsequence must also be a supersequence of a given string z [2,1,25]. This problem itself is interesting from the combinatorial point of view but its main motivation comes from bioinformatics: in the computation of the homology of two biological sequences it is important to take into account a common specific or putative structure [25]. This problem was introduced, quite recently, by Tsai in [25], where an algorithm was presented solving the problem in $\mathcal{O}(n^4 r)$ time complexity. Here $r = |z|$. Later, Chin et al. [6] and independently, Arslan and Eğecioğlu [2,1] presented improved algorithm with $\mathcal{O}(n^2 r)$ time and space complexity. Very recently, Iliopoulos and Rahman [11] presented another efficient algorithm for CLCS problem having time complexity $O(\mathcal{R} r \log\log n)$, where $\mathcal{R}$ is the number of ordered matches between the two (main) strings. To the best of our knowledge, there has not been any work on the general version of the CLCS problem, i.e., the k-CLCS problem in the literature.

In this paper, we show how the k-LCS, k-SCS and k-CLCS problems for degenerate strings can be solved efficiently using the two finite automata *SubAtm* and *SuperAtm*. The motivation to consider degenerate strings comes from the fact that these are extensively used in molecular biology to express polymorphism in DNA sequences, e.g. the polymorphism of protein coding regions caused by redundancy of the genetic code or polymorphism in binding site sequences of a family of genes. To the best of our knowledge, the only work that directly relates to our research is the very recent paper of Iliopoulos et al. [12], where LCS and CLCS problems on degenerate strings were considered and a number of efficient algorithms were presented. However, in [12], the authors only consider the two-strings version of both the problems and do not consider the SCS problem. Furthermore, to achieve efficiency, the authors in [12] assumes that the sets of letters in the degenerate strings are given in sorted order.

The rest of the paper is organized as follows. In Section 2, we present the notations and definitions. In Sections 3 and 4 we present our main results. In particular, we present the construction algorithms for *SubAtm* and *SuperAtm* in Section 3 and in Section 4, we show how they can be used to solve LCS, SCS and CLCS for degenerate strings. Finally, we briefly conclude in Section 5.

2. Preliminaries

An *alphabet* Σ is a non-empty finite set of symbols. A *string* over a given alphabet is a finite sequence of symbols. The *empty string* is denoted by ε. The set of all strings over an alphabet Σ (including empty string ε) is denoted by Σ^*.

Definition 1. A string $\tilde{x} = \tilde{x}_1\tilde{x}_2\ldots\tilde{x}_n$ is said to be degenerate, if it is built over the potential $2^{|\Sigma|} - 1$ non-empty sets of letters belonging to Σ. We say that $\tilde{x}_i \in \{\mathcal{P}(\Sigma) \setminus \emptyset\} = \mathcal{P}^+(\Sigma)$ and $|\tilde{x}_i|$ denotes the cardinality of $\tilde{x}_i$. The empty degenerate string is denoted $\tilde{\varepsilon}$.

In what follows, the set containing the letters a and c will be denoted by $[a, c]$ and the singleton $[c]$ will be simply denoted by c for the ease of reading. Also, we use the following convention: we use normal letters like x to denote normal strings. The degenerate strings are denoted by normal letters with tilde accent, e.g. $\tilde{x}$.

Definition 2. The *Language represented by degenerate string* $\tilde{x}$ is the set $\mathcal{L}(\tilde{x}) = \{u \mid u = u_1u_2\ldots u_n,\ u_j \in \tilde{x}_j,\ 1 \leqslant j \leqslant n, u \in \Sigma^*\}$. A string u is said to be an *element* of degenerate string $\tilde{x}$, denoted $u \in \tilde{x}$, if it is an element of language represented by $\tilde{x}$. The language represented by $\tilde{\varepsilon}$ is $\mathcal{L}(\tilde{\varepsilon}) = \{\varepsilon\}$.

We define the concatenation operation on the set of degenerate strings in the usual way: if $\tilde{x}$ and $\tilde{y}$ are degenerate strings over Σ, then the concatenation of these strings is $\tilde{x}\tilde{y}$. The length $|\tilde{x}|$ of degenerate string $\tilde{x}$ is the number of its sets (of letters). Note that, there may exist singleton sets. A degenerate string $\tilde{v}$ is a *factor* (resp. *prefix, suffix*) of a degenerate string $\tilde{x}$ if $\tilde{x}=\tilde{u}\tilde{v}\tilde{w}$ (resp. $\tilde{x}=\tilde{v}\tilde{w}$, $\tilde{x}=\tilde{u}\tilde{v}$).

Definition 3. A (normal) string $v=v_1\dots v_\ell$ of length ℓ is said to *occur* in a degenerate string $\tilde{x}=\tilde{x}_1\tilde{x}_2\dots\tilde{x}_n$ at position i, $1\leqslant i\leqslant n$, if, and only if, $v_j\in\tilde{x}_{i-\ell+j}$, for all $1\leqslant j\leqslant\ell$. The position of the *first occurrence* of v in $\tilde{x}$ is denoted by $foccur(v,\tilde{x})$.

Definition 4 *(Sets of Subsequences of Degenerate Strings).* For a set of $k\geqslant 2$ degenerate strings, $S=\{\tilde{x}_1,\tilde{x}_2,\dots,\tilde{x}_k\}$ and degenerate string $\tilde{z}$ over an alphabet Σ, where $\tilde{x}_j=\tilde{x}_{j1}\tilde{x}_{j2}\dots\tilde{x}_{jn_j}$ for $1\leqslant j\leqslant k$, we define the followings sets:

(i) *Set of all subsequences* of $\tilde{x}_j$:

$$\begin{aligned}Sub(\tilde{x}_j)=\Big\{\,u\in\Sigma^*\mid u\in\tilde{x}_{ji_1}\tilde{x}_{ji_2}\cdots\tilde{x}_{ji_m}\wedge\tilde{x}_j=&\,\tilde{x}_{j1}\tilde{x}_{j2}\cdots\tilde{x}_{ji_1-1}\tilde{x}_{ji_1}\\&\,\tilde{x}_{ji_1+1}\cdots\tilde{x}_{ji_2-1}\tilde{x}_{ji_2}\\&\,\tilde{x}_{ji_2+1}\cdots\tilde{x}_{ji_m-1}\tilde{x}_{ji_m}\\&\,\tilde{x}_{ji_m+1}\cdots\tilde{x}_{jn},\\\tilde{x}_{ji}\in\mathcal{P}^+(\Sigma),\ 1\leqslant i\leqslant n,\ 1\leqslant i_1<i_2<\cdots<i_m\leqslant n,\ m\geqslant 0\Big\}.&\end{aligned}$$

(ii) *Set of all common subsequences* of S:

$$CSub(S)=\left\{u\in\Sigma^*\ \Bigg|\ \bigwedge_{j=1}^{k}u\in Sub(\tilde{x}_j)\right\}.$$

(iii) *Set of all longest common subsequences* of S:

$$LCSub(S)=\big\{u\in\Sigma^*\mid u\in CSub(S)\wedge\forall v\in CSub(S):\ |v|\leqslant|u|\big\}.$$

(iv) *Set of all constrained common subsequences* of S with respect to $\tilde{z}$:

$$CCSub(S,\tilde{z})=\big\{u\in\Sigma^*\mid u\in CSub(S)\wedge\exists v:\ v\in\tilde{z}\wedge v\in Sub(u)\big\}.$$

(v) *Set of all constrained longest common subsequences* of S with respect to $\tilde{z}$:

$$CLCSub(S,\tilde{z})=\big\{u\in\Sigma^*\mid u\in CCSub(S,\tilde{z})\wedge\forall v\in CCSub(S,\tilde{z}):\ |v|\leqslant|u|\big\}.$$

Definition 5 *(Sets of Supersequences of Degenerate Strings).* For a set of $k\geqslant 2$ degenerate strings, $S=\{\tilde{x}_1,\tilde{x}_2,\dots,\tilde{x}_k\}$, over an alphabet Σ, where $\tilde{x}_j=\tilde{x}_{j1}\tilde{x}_{j2}\dots\tilde{x}_{jn_j}$ for $2\leqslant j\leqslant k$, we define followings sets:

(i) *Set of all supersequences* of $\tilde{x}_j$:

$$Super(\tilde{x}_j)=\big\{v\in\Sigma^*\mid\exists\,u\in\tilde{x}_j:\ u\in Sub(v)\big\}.$$

(ii) *Set of all common supersequences* of S:

$$CSuper(S)=\left\{u\in\Sigma^*\ \Bigg|\ \bigwedge_{j=1}^{k}u\in Super\big(\tilde{x}_j\big)\right\}.$$

(iii) *Set of all shortest common supersequences* of S:

$$SCSuper(S)=\big\{u\in\Sigma^*\mid u\in CSuper(S)\wedge\forall v\in CSuper(S):\ |v|\geqslant|u|\big\}.$$

Definition 6 *(Maximum Length Function, Minimum Length Function).* Let L be a language over an alphabet Σ.

(i) The *Maximum Length Function*, $MaxLen:\mathcal{P}(\Sigma^*)\mapsto\mathcal{P}(\Sigma^*)$, is defined for L as

$$MaxLen(L)=\big\{w\mid w\in L\wedge\forall v\in L:\ |v|\leqslant|w|\big\}.$$

(ii) The *Minimum Length Function*, $MinLen:\mathcal{P}(\Sigma^*)\mapsto\mathcal{P}(\Sigma^*)$, is defined for L as

$$MinLen(L)=\big\{w\mid w\in L\wedge\forall v\in L:\ |v|\geqslant|w|\big\}.$$

We now give brief definitions to the related concepts of finite automata. A nondeterministic finite automaton M is a quintuple $(Q, \Sigma, \delta, I, F)$, where: Q is a finite set of states, Σ is an input alphabet, δ is a mapping $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \mapsto \mathcal{P}(Q)$ called a state transition function, $I \subseteq Q$ is a set of initial states, and $F \subseteq Q$ is a set of final states. A deterministic finite automaton M is a special case of nondeterministic finite automaton, such that, transition mapping is a function δ: $Q \times \Sigma \mapsto Q$ and there is only one initial state $q_0 \in Q$. The *left language* of state q of finite automaton M, denoted $\overleftarrow{\mathcal{L}}_M(q)$, is a set of strings, for which there exists a sequence of transitions from the initial state to state q. *Language accepted* by finite automaton M, denoted $\mathcal{L}(M)$, is a set of words, for which there exists a sequence of transitions from the initial state to some of the final states. The *depth* of state q of acyclic finite automaton M is the length of the longest string in its left language.

We conclude this section by giving a formal definition of the LCS, the SCS and the CLCS problem for degenerate strings along with an example.

Problem "DLCS" (LCS Problem for Degenerate Strings) 1. *Given a set of $k \geqslant 2$ degenerate strings $S = \{\tilde{x}_1, \tilde{x}_2, \ldots, \tilde{x}_k\}$, over an alphabet Σ, the DLCS problem for S is to compute the set $LCSub(S)$.*

Problem "DSCS" (SCS Problem for Degenerate Strings) 1. *Given set of $k \geqslant 2$ degenerate strings $S = \{\tilde{x}_1, \tilde{x}_2, \ldots, \tilde{x}_k\}$, over an alphabet Σ, the DSCS problem for S is to compute the set $SCSuper(S)$.*

Problem "DCLCS" (CLCS Problem for Degenerate Strings) 1. *Given set of $k \geqslant 2$ degenerate strings $S = \{\tilde{x}_1, \tilde{x}_2, \ldots, \tilde{x}_k\}$ and degenerate string $\tilde{z}$ over an alphabet Σ, the DCLCS problem for S and $\tilde{z}$ is to compute the set $CLCSub(S, \tilde{z})$.*

Example 7. Suppose, we are given the sets of degenerate strings $S_1 = \{\tilde{x}_1 = aba[b,c], \tilde{x}_2 = abb[a,c]\}$ and $S_2 = \{\tilde{y}_1 = b[a,c], \tilde{y}_2 = [a,c]b\}$. The table on the left shows an example of an LCS of S_1 and the table on the right shows an example of an SCS of S_2.

$\tilde{x}_1$	a	b	a	$[b,c]$	
$\tilde{x}_2$	a	b		b	$[a,c]$
LCS	a	b		b	

$\tilde{y}_1$		b	$[a,c]$
$\tilde{y}_2$	$[a,c]$	b	
SCS	a	b	a

Example 8. Suppose, we are given set of degenerate strings $S_3 = \{\tilde{x} = [a,f]bddaaa,\ \tilde{y} = [a,c]ba[c,d]aa[d,f]\}$ and degenerate string $\tilde{z} = b[c,d]d$. The table on the left shows an example of LCS of S_3 and the table on the right shows an example of CLCS of S_3 with respect to degenerate string $\tilde{z}$. Note that although the LCS is of length 5 the CLCS is of length 4 only.

$\tilde{x}$	$[a,f]$	b		d	d	a	a	a
$\tilde{y}$	$[a,c]$	b	a	$[c,d]$		a	a	$[d,f]$
LCS	a	b		d		a	a	

$\tilde{x}$	$[a,f]$	b		d			d	a	a	a
$\tilde{y}$	$[a,c]$	b	a	$[c,d]$	a	a	$[d,f]$			
$\tilde{z}$		b		$[c,d]$			d			
$CLCS$	a	b		d			d			

3. Subsequence and supersequence automata for degenerate strings

In this section, we present two novel types of finite automata, namely the *subsequence automaton* (*SubAtm*), which accepts the set of all subsequences of a given degenerate string and the *supersequence automaton* (*SuperAtm*), which accepts the set of all supersequences of a given degenerate string. Both *SubAtm* and *SuperAtm* are deterministic and minimal and its number of states and transitions are linear with respect to the length of the given degenerate strings. Moreover, *SubAtm* is acyclic. Next, we present online linear time algorithms to construct *SubAtm* and *SuperAtm*.

The set of all subsequences and (resp. supersequences) of a given degenerate string is a regular language and therefore, according to basic principles of finite formal language theory [10], it is possible to construct a finite automaton accepting the set. The considered sets can be expressed by simple regular expressions, which can be easily transformed by standard procedure [10] into nondeterministic finite automata with linear number of states with respect to the length of the input strings. For our case however, we need deterministic finite automata instead of the nondeterministic ones. These can be obtained by determinization of NFAs. However, after the determinization, the resulting automaton, in general, can have up to 2^n states, not to mention the unacceptable time requirements of the determinization process.

3.1. Subsequence automaton

The subsequence automaton, *SubAtm* for degenerate string $\tilde{x} = \tilde{x}_1\tilde{x}_2 \ldots \tilde{x}_n$, can be defined as $M = (Q, \Sigma, \delta, q_0, F)$, where $Q = F = \{q_0, q_1, q_2, \ldots, q_n\}$ and $\delta(q_i, s) = q_j,\ j = \min\{k \mid s \in \tilde{x}_k,\ i + 1 \leqslant k \leqslant n\}$ if $\{k \mid s \in \tilde{x}_k,\ i + 1 \leqslant k \leqslant n\} \neq \emptyset$; otherwise $\delta(q_i, s)$ is undefined. For an example of *SubAtm*, see Fig. 1 (automata M_1, M_2).

Next, we present an online algorithm to compute *SubAtm*. The algorithm builds *SubAtm* from left to right, i.e., in each step, $SubAtm(\tilde{x}_1\tilde{x}_2 \ldots \tilde{x}_k)$ is extended to $SubAtm(\tilde{x}_1\tilde{x}_2 \ldots \tilde{x}_{k+1})$. This operation is performed by adding a new state q_{k+1}, on

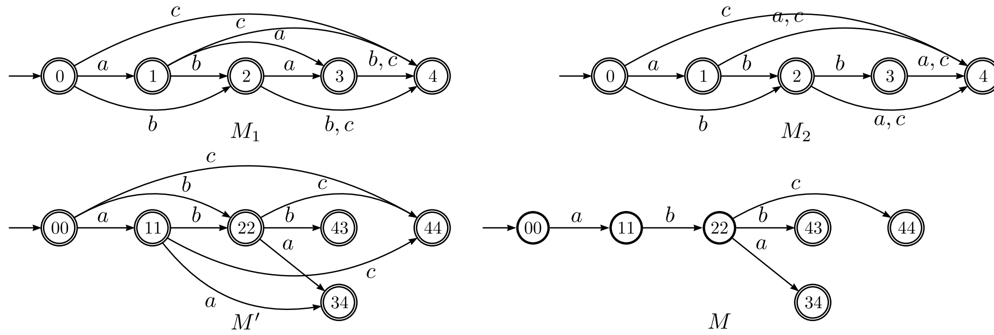

Fig. 1. Deterministic finite automata M_1, M_2, M' and M, where $\mathcal{L}(M_1) = \mathit{Sub}(\tilde{x})$, $\mathcal{L}(M_2) = \mathit{Sub}(\tilde{y})$, $\mathcal{L}(M_1) = \mathcal{L}(M_1) \cap \mathcal{L}(M_2)$, $\mathcal{L}(M) = \mathit{MaxLen}(\mathcal{L}(M')) = \mathit{LCSub}(\tilde{x}, \tilde{y}) = \{abc, abb, aba\}$, for $\tilde{x} = aba[b, c]$ and $\tilde{y} = abb[a, c]$.

the left of the automaton and then, by adding new transitions for all symbols $s \in \tilde{x}_{k+1}$ leading from all states q_ℓ not having transition for s, where $0 \leqslant \ell \leqslant k$. The new transitions are added as follows. Firstly, let us consider a prefix $\tilde{x}[1..k] = \tilde{x}_1\tilde{x}_2 \ldots \tilde{x}_i\tilde{x}_{i+1} \ldots \tilde{x}_k$, such that, $s \in \tilde{x}_i$ and $s \notin \tilde{x}_l$, where $1 \leqslant i < l \leqslant k < n$. Then, it holds that, $\delta(q_l, s)$ is undefined for $i \leqslant l \leqslant k$. Secondly, let us consider extended prefix $\tilde{x}[1..k+1] = \tilde{x}_1\tilde{x}_2 \ldots \tilde{x}_i\tilde{x}_{i+1} \ldots \tilde{x}_k\tilde{x}_{k+1}$, where $\tilde{x}_{k+1}$ is such that $s \in \tilde{x}_{k+1}$. Then it holds that $\delta(q_l, s) = q_{k+1}$ for $i \leqslant l \leqslant k$. Therefore, for the added symbol s, it is sufficient to add a transition only to the states $q_i, q_{i+1}, \ldots, q_k$. This can be easily ensured by memorizing the leftmost state (its index) for each symbol of the alphabet not having the appropriate transition. The steps are formally presented in Algorithm 1.

procedure: SUB-ATM
input: degenerate string $\tilde{x} = \tilde{x}_1\tilde{x}_2 \ldots \tilde{x}_n$
output: subsequence automaton M, $\mathcal{L}(M) = \mathit{Sub}(\tilde{x})$

1 **begin**
2 $\quad Q \leftarrow \{q_0\}$; $\delta \leftarrow \emptyset$
3 $\quad$ **for each** $s \in \Sigma$ **do** $\mathit{leftm}[s] \leftarrow 0$
4 $\quad$ **for** $i \leftarrow 1$ **to** n **do**
5 $\quad\quad Q \leftarrow Q \cup \{q_i\}$
6 $\quad\quad$ **for each** $s \in \tilde{x}_i$ **do**
7 $\quad\quad\quad$ **for** $j \leftarrow \mathit{leftm}[s]$ **to** $i - 1$ **do** $\delta(q_j, s) \leftarrow q_i$
8 $\quad\quad\quad \mathit{leftm}[s] \leftarrow i$
9 $\quad\quad$ **end for**
10 $\quad$ **end for**
11 $\quad M \leftarrow (Q, \Sigma, \delta, q_0, Q)$
12 $\quad$ **return** M
13 **end**

Algorithm 1. Online construction of *SubAtm*.

The basic properties of the subsequence automaton and of Algorithm 1 are summarized in the following lemmas.

Lemma 9. *Let $M = (Q, \Sigma, \delta, q_0, F)$ be automaton output by Algorithm 1, given the degenerate string $\tilde{x} = \tilde{x}_1\tilde{x}_2 \ldots \tilde{x}_n$ as the input. Then, for all $q \in Q$ and $u \in \Sigma^*$, it holds that*

$$\big(u \in \overleftarrow{\mathcal{L}}_M(q) \wedge \mathit{depth}(q) = i\big) \quad \Leftrightarrow \quad \big(u \in \mathit{Sub}(\tilde{x}) \wedge \mathit{foccur}(u, \tilde{x}) = i\big).$$

Proof. We first prove the “if” part by applying induction on the number of states. (Note that $\mathit{depth}(q_i) = i$.)

(i) *Base case*: $i = 0$. The construction gives $\overleftarrow{\mathcal{L}}_M(q_0) = \{\varepsilon\}$ and by the definition of $\mathit{Sub}(\tilde{x})$, it holds $\varepsilon \in \mathit{Sub}(\tilde{x})$.

(ii) *Induction step*: Let us assume that the assertion holds for all states q_i, where $0 \leqslant i \leqslant k-1$. We show that it holds for q_k as well. Now, consider a $u \in \overleftarrow{\mathcal{L}}_M(q_k)$. Then, there exists a sequence of transitions $\delta(q_0, u_1) = q_{j_1}$, $\delta(q_{j_1}, u_2) = q_{j_2}, \ldots, \delta(q_{j_{m-1}}, u_m) = q_{j_m} = q_k$. The construction gives $u_m \in \tilde{x}_k$ and by the hypothesis, we have $u_1, \ldots, u_{m-1} \in \mathit{Sub}(\tilde{x})$ and $\mathit{foccur}(u_1, \ldots, u_{m-1}, \tilde{x}) = j_{m-1}$. The algorithm constructs all the transitions in the way they are oriented from a state with lower number to a state with higher number, and hence $j_{m-1} < k$. Hence by the definition of subsequence we get $u_1, \ldots, u_{m-1}, u_m \in \mathit{Sub}(\tilde{x})$, u occurs at position k in $\tilde{x}$. Moreover, according to the construction, the occurrence of u_m at position k in the generalized suffix $\tilde{x}_{j_{m-1}+1}, \ldots, \tilde{x}_n$ is the first one. Hence, it holds that $\mathit{foccur}(u, \tilde{x}) = k$.

We now prove the “only if” part by induction on the length ℓ of u.

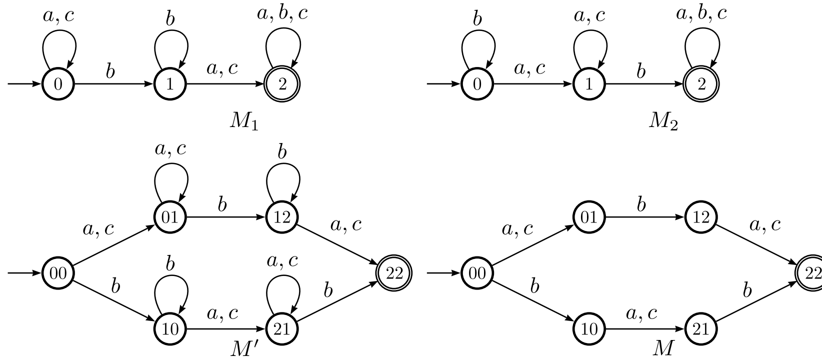

Fig. 2. Deterministic finite automata M_1, M_2, M' and M, where $\mathcal{L}(M_1) = Super(\tilde{x})$, $\mathcal{L}(M_2) = Super(\tilde{y})$, $\mathcal{L}(M') = \mathcal{L}(M_1) \cap \mathcal{L}(M_2)$, $\mathcal{L}(M) = MinLen(\mathcal{L}(M')) = SCSuper(\tilde{x}, \tilde{y}) = \{aba, abc, cba, cbc, bcb, bab\}$, for $\tilde{x} = b[a, c]$, $\tilde{y} = [a, c]b$.

(i) *Base case*: $\ell = 1$. Let us consider $u = u_1$ and $foccur(u_1, \tilde{x}) = i$. Then, there exists a transition $\delta(q_0, u_1) = q_i$, what is given by the construction. As a consequence, we have $u_1 \in \overleftarrow{\mathcal{L}}_M(q_i)$.

(ii) *Induction step*: Let us assume that the assertion holds for all strings u with length $0 \leqslant \ell \leqslant k - 1$. We now show that it holds also for $u = u_1 u_2 \ldots u_k$. Let us consider $u \in Sub(\tilde{x}) \wedge foccur(u, \tilde{x}) = i_k$. By hypothesis we have $u_1 u_2 \ldots u_{k-1} \in Sub(\tilde{x})$ and $foccur(u_1 u_2 \ldots u_{k-1}, \ \tilde{x}) = i_{k-1}$ and $u_1 u_2 \ldots u_{k-1} \in \overleftarrow{\mathcal{L}}_M(q_{i_{k-1}})$. By the definition of *foccur*, it holds that $i_{k-1} < i_k$. Furthermore we know that there is a first occurrence of u_k at position i_k in the suffix $\tilde{x}_{i_{k-1}+1}, \ldots, \tilde{x}_n$. But from the construction, there exists a transition $\delta(q_{i_{k-1}}, u_k) = q_{i_k}$. Therefore, it holds $\overleftarrow{\mathcal{L}}_M(q_{i_{k-1}}) \cdot \{u_k\} \subseteq \overleftarrow{\mathcal{L}}_M(q_{i_k})$ and finally $u \in \overleftarrow{\mathcal{L}}_M(q_{i_k})$. □

Lemma 10. *Given a degenerate string $\tilde{x}$ of length n over an alphabet Σ, Algorithm 1 correctly constructs deterministic finite automaton M accepting language $\mathcal{L}(M) = Sub(\tilde{x})$ in $\mathcal{O}(n|\Sigma|)$ time using $\mathcal{O}(|\Sigma|)$ additional space. M is acyclic, minimal and has exactly $n + 1$ states and $\mathcal{O}(|\Sigma|n)$ transitions.*

Proof. Since all the states of M are final, by Lemma 9, we must have $\mathcal{L}(M) = Sub(\tilde{x})$.

Now, let us consider the number of states and transitions of M. The algorithm first creates the initial state and then it creates one new state in each step. The total number of steps is given by the length n of x. Hence the total number of states is $n + 1$. Now, M can get the maximum number of transitions only when all symbol-sets of $\tilde{x}$ are equal to σ. Each step of the algorithm adds exactly $|\sigma|$ transitions leading from the current last state to the new added state of the automaton.

Next, we prove that M is deterministic. The values of the array $leftm[s]$ for a given s divide the interval $0..n$ of the numbers of states into disjoint subintervals. Each state with the number within the subinterval is visited and a transition for s is added exactly once. Therefore, M remains deterministic.

Now we prove that M is acyclic and minimal. Clearly M is acyclic because the algorithm adds transitions to states only in the way that a transition leads from a state with a lower number to a state with a higher number.

The longest subsequence of $\tilde{x}$ is consisted of a letter from each of its sets (of letters). Therefore, in the automaton, there must exists a sequence of transitions of length n. Now since the automaton is acyclic, the automaton has to have at least $n + 1$ states. Therefore, since we have already proven that M has exactly $n + 1$ states, M must be minimal.

Finally, we concentrate on the time and space complexities of the algorithm. One iteration of the outer loop beginning at line 4 adds one state and certain number of transition. The loop is performed n times. However, as we have already established, the total number of transitions is in the worst case $n|\Sigma|$ (see above). Hence the total time complexity is $\mathcal{O}(n|\Sigma|)$.

The additional space requirement is only due to the array $leftm[s]$ of size $|\Sigma|$. Hence, the space complexity of the algorithm is $O(|\Sigma|)$. □

3.2. Supersequence automaton

The supersequence automaton, *SuperAtm* for degenerate string $\tilde{x} = \tilde{x}_1 \tilde{x}_2 \ldots \tilde{x}$ can be defined as $M = (Q, \Sigma, \delta, q_0, F)$, where $Q = \{q_0, q_1, q_2, \ldots, q_n\}$, $F = \{q_n\}$ and $\delta(q_{i-1}, s) = q_i$, for all $s \in \tilde{x}_i$ and $\delta(q_{i-1}, s) = q_{i-1}$, otherwise, where $1 \leqslant i \leqslant n$, and $\delta(q_n, s) = q_n$ for all $s \in \Sigma$. For an example of *SuperAtm*, see Fig. 2 (automata M_1, M_2).

The algorithm for the online construction of *SuperAtm* is presented in Algorithm 2. It is easy to realize that Algorithm 2 is a straightforward realization of the definition of *SuperAtm*.

```
procedure: SUPER-ATM
input: a degenerate string x̃ = x̃_1 x̃_2 ... x̃_n
output: supersequence automaton M, L(M) = Super(x̃)
1  begin
2      Q ← {q_0}; last ← q_0
3      for i ← 1 to n do
4          Q ← Q ∪ {q_i}
5          last ← q_i
6          for each s ∈ x̃_i do δ(q_{i−1}, s) ← q_i
7          for each s ∈ {Σ \ x̃_i} do δ(q_{i−1}, s) ← q_{i−1}
8      end for
9      for each s ∈ Σ do δ(q_n, s) ← q_n
10     M ← (Q, Σ, δ, q_0, {q_n})
11     return M
12 end
```

Algorithm 2. Construction of *SuperAtm*.

The basic properties of the *SuperAtm* and Algorithm 2 are summarized in the following lemma.

Lemma 11. *Given a degenerate string $\tilde{x}$ of length n, Algorithm 2 correctly constructs a deterministic finite automaton M accepting language $\mathcal{L}(M) = Super(\tilde{x})$ in $\mathcal{O}(|\Sigma|n)$ time. M has $n+1$ states and $|\Sigma|(n+1)$ transitions.*

Proof. First, we prove correctness of the algorithm. Then we prove the time and space complexities of the algorithm before considering the size of the resulting automaton.

(i) *Correctness*

Clearly it is sufficient to prove that $u \in \mathcal{L}(M) \Leftrightarrow u \in Super(\tilde{x})$, for all $u \in \Sigma, \tilde{x} \in \mathcal{P}^+(\Sigma)^*$.
First, we define automata M and M', both based on the degenerate string $\tilde{x} = \tilde{x}_2 \ldots \tilde{x}_n$. M is defined as a quintuple $M = (Q, \Sigma, \delta, q_1, F)$, where $Q = \{q_1, \ldots, q_n\}$, $F = \{q_n\}$ and $\delta = \{\forall i \in [2..n]$: $(\forall s \in \Sigma \setminus \tilde{x}_i : \delta(q_{i-1}, s) = q_{i-1}, \forall s \in \tilde{x}_i : \delta(q_{i-1}, s) = q_i)$, $\forall s \in \Sigma : \delta(q_n, s) = q_n$.
M', on the other hand, is defined as a quintuple, $M' = (Q', \Sigma, \delta', q_0, F)$, where $Q' = Q \cup \{q_0\}$ and $\delta' = \delta \cup \{\forall s \in \Sigma \setminus \tilde{x}_1\colon\ \delta(q_0, s) = q_0, \forall s \in \tilde{x}_1\colon\ \delta(q_0, s) = q_1\}$.
We further define automaton M'' constructed for $\tilde{x} = \tilde{\varepsilon}$, as $M'' = (Q'', \Sigma, \delta, q_0, F')$, where $Q'' = \{q_0\}$, $F'' = \{q_0\}$ and $\delta'' = \{\forall s \in \Sigma\colon\ \delta(q_0, s) = q_0\}$. Now we are ready to prove the correctness of the lemma. We first prove the "if" part and then we consider the "only if" part. Both the parts are proved by induction on the length ℓ of $\tilde{x}$.
"$\Rightarrow$"

(a) *Base case*: $\ell = 0$. We need to consider string $\tilde{x} = \tilde{\varepsilon}$ and hence the automaton M''. By construction we have $\mathcal{L}(M'') = \Sigma^*$ and by Definition 4, we have $Super(\tilde{\varepsilon}) = \Sigma^*$. Hence $\mathcal{L}(M'') = Super(\tilde{\varepsilon})$.
(b) *Induction step*: We assume it holds that $u \in \mathcal{L}(M) \Rightarrow u \in Super(\tilde{x})$, for all $u \in \Sigma, \tilde{x} \in \mathcal{P}^+(\Sigma)^*$, where $|\tilde{x}| < n$. Now, our strategy is to consider string $\tilde{x}_2 \ldots \tilde{x}_n$ and automaton M for which the hypothesis holds and, using that, we will show that it also holds for string $\tilde{x}_1 \ldots \tilde{x}_n$ and automaton M'. By the induction hypothesis, M accepts strings of the form $u = w_1 s_2 w_2 s_3 \ldots s_n w_n$, where $s_i \in \tilde{x}_i$, $w_j \in \Sigma^*$. By the construction, it holds that $\mathcal{L}(M') = (\Sigma \setminus \tilde{x}_i)^* \tilde{x}_1 \mathcal{L}(M)$. Hence the string accepted by M' is of the form $u' = v_0 s_1 u$, i.e., $u' = v_0 s_1 w_1 s_2 v_2 s_3 \ldots s_n w_n$, where $v_0 \in (\Sigma \setminus \tilde{x}_1)^*$, $s_1 \in \tilde{x}_1$. Therefore, we have $u \in Super(\tilde{x}_1 \ldots \tilde{x}_n)$.

"$\Leftarrow$"

(a) *Base case*: Similar to the "if" part.
(b) *Induction step*: We assume it holds that $u \in Super(\tilde{x}) \Rightarrow u \in \mathcal{L}(M)$, for all $u \in \Sigma, \tilde{x} \in \mathcal{P}^+(\Sigma)^*$, where $|\tilde{x}| < n$. As above, our strategy is to consider string $\tilde{x}_2 \ldots \tilde{x}_n$ and automaton M for which the proposition holds and, using that, we will show that it also holds for string $\tilde{x}_1 \ldots \tilde{x}_n$ and automaton M'. Let us have string $u \in Super(\tilde{x}_1 \ldots \tilde{x}_n)$ and we can consider w.l.o.g. the string is of the form $u' = v_0 s_1 v_1 s_2 \ldots s_n w_n$, where $v_{i-1} \in (\Sigma \setminus \tilde{x}_i)^*$ $w_n \in \Sigma^*$ and $s_i \in \tilde{x}_i$, so we consider the first occurrence of $s_1 \ldots s_n$ in u. Since by the construction we have $\mathcal{L}(M') = (\Sigma \setminus \tilde{x}_i)^* \tilde{x}_1 \mathcal{L}(M)$ and since according to induction preposition the string $u = v_1 s_2 \ldots v_n w_n$ is accepted by automaton M, it must hold that the string $u' = v_0 s_1 u = v_0 s_1 v_1 s_2 \ldots s_n w_n$ is accepted by M'.

(ii) *Time complexity*

The loop at line 3 is performed n times and the total number of transitions created in a single iteration (either in loop at line 6 or in loop at line 7) is exactly $|\Sigma|$. Hence the total time complexity is $\mathcal{O}(|\Sigma|n)$.

(iii) *Size of the resulting automaton*:

By the construction, one state are created during initialization and n states are created by the for loop at line 3. Exactly $n|\Sigma|$ transitions in total are created the by for loops at line 6 and at line 7. Another n transitions are created by for loop at line 9. Hence the resulting automaton has $n+1$ states and $(n+1)|\Sigma|$ transitions. □

4. Algorithms for LCS, SCS and CLCS

In this section, we show how we can use *SubAtm* and *SuperAtm* to efficiently solve Problems DLCS, DSCS and DCLCS. Both the algorithms to solve Problems DLCS and DSCS work in $\mathcal{O}(|\Sigma|\prod_{j=1}^{k} n_j)$ time and space. The algorithm to solve Problem DCLCS on the other hand works in $\mathcal{O}(|\Sigma|(\prod_{j=1}^{k} n_j)r)$ time and space. In what follows, we discuss all the algorithms simultaneously, because, they basically follow a common scheme, differing only in particular steps. The algorithms are based on Lemmas 12, 13 and 14, respectively.

Lemma 12. *Let $S = \{\tilde{x}_1, \tilde{x}_2, \ldots, \tilde{x}_k\}$ be a set of degenerate strings over an alphabet Σ, then the following holds*:

$$LCSub(S) = MaxLen\left(\bigcap_{j=1}^{k} Sub(\tilde{x}_j)\right).$$

Lemma 13. *Let $S = \{\tilde{x}_1, \tilde{x}_2, \ldots, \tilde{x}_k\}$ be a set of degenerate strings over an alphabet Σ, then the following holds*:

$$SCSuper(S) = MinLen\left(\bigcap_{j=1}^{k} Super(\tilde{x}_j)\right).$$

Lemma 14. *Let $S = \{\tilde{x}_1, \tilde{x}_2, \ldots, \tilde{x}_k\}$ be a set of degenerate strings and $\tilde{z}$ be degenerate string over an alphabet Σ, then the following holds*:

$$CLCSub(S, \tilde{z}) = MaxLen\left(\left(\bigcap_{i=1}^{k} Sub(\tilde{x}_i)\right) \cap Super(\tilde{z})\right).$$

The proof of Lemmas 12, 13 and 14 follows directly from the definitions of the sets involved. Note that, all of these sets form regular languages. This implies that we can represent each of them by a finite automaton. It is therefore sufficient to construct a finite automaton accepting the set $LCSub(S)$ (resp. $SCSuper(S)$, $CLCSub(S, r)$) to solve DLCS (resp. DSCS, DCLCS) problem. All the strings of the (output) set can then be easily obtained by finding paths (e.g. by DFS traversal) in the transition diagram of the (output) automaton from the initial state to all the final states. The string(s) are then given by spelling the labels of transitions on the paths.

The Algorithm LCSub-Atm constructing a finite automaton $M = (Q, \Sigma, \delta, q_0, F)$, such that $\mathcal{L}(M) = LCSub(S)$ for a set of degenerate strings $S = \{\tilde{x}_1, \tilde{x}_2, \ldots, \tilde{x}_k\}$, works in the following steps:

(i) For each degenerate string $\tilde{x}_j$ in S, construct automaton M_j,

$$M_j \leftarrow \text{Sub-Atm}(\tilde{x}_j), \quad \mathcal{L}(M_j) = Sub(\tilde{x}_j).$$

(ii) From automata $M_1, M_2, \ldots, M_k$ construct automaton M',

$$M' \leftarrow \text{Intersection}(M_1, M_2, \ldots, M_k), \quad \mathcal{L}(M') = \bigcap_{j=1}^{k} \mathcal{L}(M_j) = \bigcap_{j=1}^{k} Sub(\tilde{x}_j).$$

(iii) Transform automaton M' into automaton M,

$$M \leftarrow \text{MaxLen-Atm}(M'), \quad \mathcal{L}(M) = MaxLen(\mathcal{L}(M')) = MaxLen\left(\bigcap_{j=1}^{k} Sub(\tilde{x}_j)\right).$$

An example of Algorithm LCSub-Atm is depicted in Fig. 1.

The Algorithm SCSuper-Atm constructing a finite automaton $M = (Q, \Sigma, \delta, q_0, F)$, such that $\mathcal{L}(M) = SCSuper(S)$ for a set of degenerate strings $S = \{\tilde{x}_1, \tilde{x}_2, \ldots, \tilde{x}_k\}$, works in the following steps:

(i) For each degenerate string $\tilde{x}_j$ in S, construct automaton M_j,

$$M_j \leftarrow \text{Super-Atm}(\tilde{x}_j), \quad \mathcal{L}(M_j) = Super(\tilde{x}_j).$$

(ii) From automata $M_1, M_2, \ldots, M_k$ construct automaton M',

$$M' \leftarrow \text{Intersection}(M_1, M_2, \ldots, M_k), \quad \mathcal{L}(M') = \bigcap_{j=1}^{k} \mathcal{L}(M_j) = \bigcap_{j=1}^{k} Super(\tilde{x}_j).$$

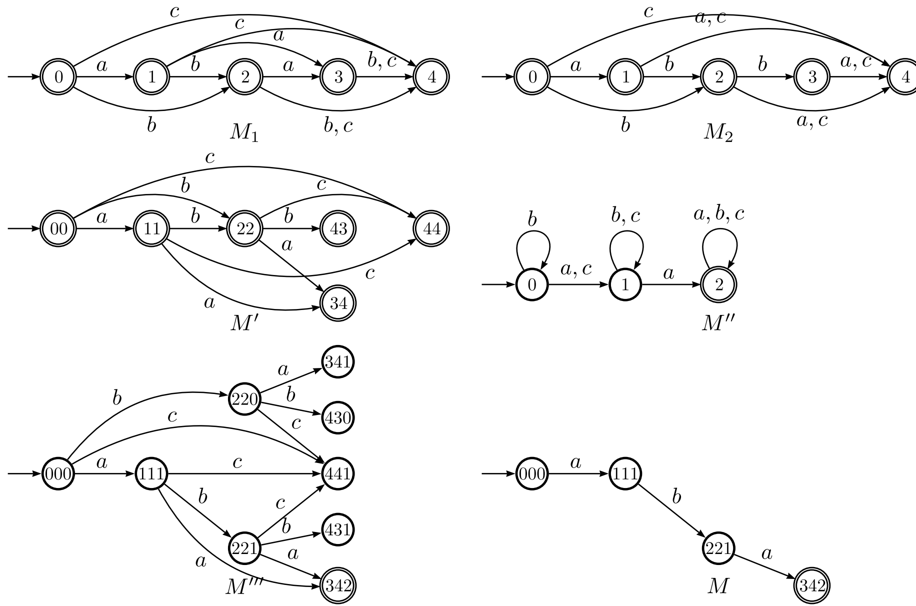

Fig. 3. Deterministic finite automata M_1, M_2, M', M'', M''' and M, where $\mathcal{L}(M_1) = Sub(\tilde{x})$, $\mathcal{L}(M_2) = Sub(\tilde{x})$, $\mathcal{L}(M') = \mathcal{L}(M_1) \cap \mathcal{L}(M_2)$, $\mathcal{L}(M'') = Super(\tilde{z})$, $\mathcal{L}(M''') = \mathcal{L}(M') \cap \mathcal{L}(M'')$, $\mathcal{L}(M) = MaxLen(\mathcal{L}(M''')) = CLCSub(\tilde{x}, \tilde{y}, \tilde{z}) = \{aba\}$, for $\tilde{x} = aba[b,c]$, $\tilde{y} = abb[a,c]$ and $\tilde{z} = [a,c][a]$.

(iii) Transform automaton M' into automaton M,

$$M \leftarrow \text{MinLen-Atm}(M'), \quad \mathcal{L}(M) = MinLen\big(\mathcal{L}(M')\big) = MinLen\left(\bigcap_{j=1}^{k} Super(\tilde{x}_j)\right).$$

An example of Algorithm SCSuper-Atm is depicted in Fig. 2.

The Algorithm CLCSub-Atm constructing a finite automaton $M = (Q, \Sigma, \delta, q_0, F)$, such that $\mathcal{L}(M) = CLCSub(S, \tilde{z})$ for a set of degenerate strings $S = \{\tilde{x}_1, \tilde{x}_2, \dots, \tilde{x}_k\}$ and a degenerate string $\tilde{z}$, works in the following steps:

(i) For each degenerate string $\tilde{x}_j$ in S, construct automaton M_j,

$$M_j \leftarrow \text{Sub-Atm}(\tilde{x}_j), \quad \mathcal{L}(M_j) = Sub(\tilde{x}_j).$$

(ii) From automata $M_1, M_2, \dots, M_k$ construct automaton M',

$$M' \leftarrow \text{Intersection}(M_1, M_2, \dots, M_k), \quad \mathcal{L}(M') = \bigcap_{j=1}^{k} \mathcal{L}(M_j) = \bigcap_{j=1}^{k} Sub(\tilde{x}_j).$$

(iii) For degenerate string $\tilde{z}$ construct automaton M'':

$$M'' \leftarrow \text{Super-Atm}(\tilde{z}), \quad \mathcal{L}(M'') = Super(\tilde{z}).$$

(iv) For automata M' and M'' construct automaton M''':

$$M''' \leftarrow \text{Intersection}(M', M''), \quad \mathcal{L}(M''') = \mathcal{L}(M') \cap \mathcal{L}(M'') = \left(\left(\bigcap_{i=1}^{k} Sub(\tilde{x}_i)\right) \cap Super(\tilde{z})\right).$$

(v) Transform automaton M''' into automaton M:

$$M \leftarrow \text{MaxLen-Atm}(M'''), \quad \mathcal{L}(M) = MaxLen\big(\mathcal{L}(M''')\big) = MaxLen\left(\left(\bigcap_{i=1}^{k} Sub(\tilde{x}_i)\right) \cap Super(\tilde{z})\right).$$

An example of Algorithm CLCSub-Atm is depicted in Fig. 3.

The first step of both Algorithm LCSub-Atm (to solve DLCS) and Algorithm CLCSub-Atm (to solve DCLCS) uses Algorithm 1 and the first step of Algorithm SCSuper-Atm (to solve DSCS) uses Algorithm 2. In the following subsections, we discuss the rest of the steps of the algorithms in detail and analyze the running time of the algorithms.

4.1. Finite automaton for intersection of languages

In this section, we discuss the construction of an automaton for the intersection of languages. This is required for the Step 2 of DLCS and DSCS algorithms and Steps 2 and 4 of DCLCS algorithm. We use a variant of that standard algorithm that creates only accessible states. Algorithm 3 formally presents the steps. As can be easily seen, this algorithm builds the resulting automaton by simultaneous traversal of the input automata. The key properties of the resulting automaton and the algorithm are presented in the following lemma:

Lemma 15. *Given deterministic finite automata $M_1, M_2, \ldots, M_k$, having $n_1, n_2, \ldots, n_k$ states, respectively, Algorithm 3 correctly constructs deterministic finite automaton M accepting language $\mathcal{L}(M) = \bigcap_{j=1}^{k} \mathcal{L}(M_k)$ in $\mathcal{O}(|\Sigma| \prod_{i=1}^{k} n_i)$ time. M has at most $\mathcal{O}(\prod_{i=1}^{k} n_i)$ states and at most $\mathcal{O}(|\Sigma| \prod_{i=1}^{k} n_i)$ transitions. Moreover, if some of automata M_i are acyclic, then M is also acyclic.*

The algorithm is adapted from [19], where the proof is provided.

procedure: INTERSECTION
input: deterministic finite automata $M_1, M_2, \ldots, M_k$, where $M_j = (Q^j, \Sigma, \delta^j, q_0^j, F^j)$, for $2 \leqslant j \leqslant k$
output: deterministic finite automaton $M = (Q, \Sigma, \delta, q_0, F)$, $\mathcal{L}(M) = \bigcap_{j=1}^{k} \mathcal{L}(M_k)$

1 **begin**
2 $\quad q_0 \leftarrow [q_0^1, q_0^2, \ldots, q_0^k]$
3 $\quad Q \leftarrow \{ [q_0^1, q_0^2, \ldots, q_0^k] \}$
4 $\quad$ **if** $\bigwedge_{j=1}^{k} (q_0^j \in F^j)$ **then** $F \leftarrow \{ [q_0^1, q_0^2, \ldots, q_0^k] \}$
5 $\quad$ **else** $F \leftarrow \emptyset$
6 $\quad C \leftarrow$ NEW-QUEUE()
7 $\quad$ ENQUEUE($C, [q_0^1, q_0^2, \ldots, q_0^k]$)
8 $\quad$ **while not** EMPTY(C) **do**
9 $\qquad [q^1, q^2, \ldots, q^k] \leftarrow$ DEQUEUE(C)
10 $\qquad$ **for all** $s \in \Sigma$ **do**
11 $\qquad\quad [p^1, p^2, \ldots, p^k] \leftarrow [\delta^1(q^1, s), \delta^2(q^2, s), \ldots, \delta^k(q^k, s)]$
12 $\qquad\quad \delta([q^1, q^2, \ldots, q^k], s) \leftarrow [p^1, p^2, \ldots, p^k]$
13 $\qquad\quad$ **if** $[p^1, p^2, \ldots, p^k] \notin Q$ **then**
14 $\qquad\qquad$ ENQUEUE($C, [p^1, p^2, \ldots, p^k]$)
15 $\qquad\qquad Q \leftarrow Q \cup \{ [p^1, p^2, \ldots, p^k] \}$
16 $\qquad\qquad$ **if** $\bigwedge_{j=1}^{k} (p^j \in F^j)$ **then** $F \leftarrow F \cup \{ [p^1, p^2, \ldots, p^k] \}$
17 $\qquad\quad$ **end if**
18 $\qquad$ **end for**
19 $\quad$ **end while**
20 $\quad M \leftarrow (Q, \Sigma, \delta, q_0, F)$
21 $\quad$ **return** M
22 **end**

Algorithm 3. Construction of an automaton for intersection of languages.

4.2. Maximum length strings and minimum length strings automaton

In this section, we discuss the algorithms for transforming an automaton representing a finite language into an automaton accepting only a subset of strings of that language having maximum length (Step 3 of DLCS algorithm, Step 5 of DCLCS algorithm) and minimum length (Step 3 of DSCS algorithm), respectively. The main idea of both the algorithms is to remove from the input automaton some states and some transitions to achieve that the resulting automaton accepts only the longest (resp. shortest) strings of the language of the input automaton. Hence, if the original automaton is $M' \leftarrow (Q', \Sigma, \delta', q_0, F')$, then the resulting automaton is $M \leftarrow (Q, \Sigma, \delta, q_0, F)$, where $Q \subseteq Q'$, $F \subseteq F'$ and $\delta \subseteq \delta'$. We use a variant of the longest-path (resp. shortest-path) algorithm for DLCS and DCLCS problems (resp. DSCS problem) so as to remove from the automaton, all the transitions and states, which are not part of any longest (resp. shortest) path from the initial state to some of the final states.

During the longest-path (resp. shortest-path) algorithm it is necessary to memorize, for each state, all its predecessor on the current longest (resp. shortest) path. We ensure this by adding new transitions leading from the target state to its predecessor. In other words, we construct a new automaton with the same set of states as the original automaton, where the set of transitions is replaced by a subset of transitions being part of some longest (resp. shortest) path from the initial state to some of the final states; however, here the transitions are reversed. Hence, the output of the longest-path (resp. shortest-path) algorithm is a nondeterministic automaton possibly with unreachable states accepting $\mathcal{L}(M_L)^R$ (resp. $\mathcal{L}(M_S)^R$). Therefore, the stage continues by applying, first the standard procedure [10] for removal of unreachable states and then, the standard procedure [10] for automata reversal to obtain the desired automaton.

4.2.1. Maximum length automaton

Given an acyclic deterministic finite automaton $M = (Q, \Sigma, \delta, q_0, F)$, Algorithm MaxLen-Atm constructs a finite automaton $M_M = (Q, \Sigma, \delta, q_0, F)$ such that $\mathcal{L}(M_M) = MaxLen(\mathcal{L}(M))$. The algorithm works in the following steps:

(i) Transform automaton M into automaton M_R such that $\mathcal{L}(M_R) = MaxLen(\mathcal{L}(M))^R$ using the longest-path algorithm for DAGs based on topological ordering of states.
(ii) Remove all unreachable states from automaton M_R using standard procedure and thereby obtain automaton M'_R such that $\mathcal{L}(M_R) = \mathcal{L}(M'_R)$.
(iii) Apply the standard reversal procedure on automaton M'_R and thereby obtain automaton M_M such that $\mathcal{L}(M_M) = \mathcal{L}(M'_R)^R$.

In case of *DLCS* problem (resp. DCLCS problem), the input of this stage is a deterministic automaton M_L accepting the set $\mathcal{L}(M_L) = \bigcap_{j=1}^{k} Sub(\tilde{x}_j)$ (resp. $\mathcal{L}(M_L) = (\bigcap_{j=1}^{k} Sub(\tilde{x}_j)) \cap \tilde{z}$). Such an automaton is acyclic (Lemma 10). Therefore, as the first step of the algorithm, we employ a modification of the longest-path algorithm for directed acyclic graphs (DAGs) based on the topological ordering of nodes (states) [7], which works in linear time with respect to the number of edges of the given graph. This (Algorithm 4) keeps for each state of the automaton M its input degree d_{in} (number of incoming transitions) and the current longest path to that state from the initial state $len[q]$. The algorithm uses queue C for closed states (states with zero input degree). In the beginning, the initial state q_0 of the automaton M is put into queue C. The main loop of the algorithm works as follows: a closed state p is dequeued. Target state q of each transition leading from p is visited, its input degree is decreased by 1 and the length of path from q_0 via p to q is computed. If such a path is the longest one from q_0 to q, p is set as a predecessor of q. Predecessor is memorized in the form of a reversed transition. That is why the resulting automaton of this stage accepts reversed longest strings of L. Obviously, if there exist more than one longest path to state q, it has more predecessors. If the input degree of state q is decreased to zero, it is closed and put into the queue C. The algorithm keeps a set S of states being targets of the longest paths form state q_0 in the graph. This set is being realized during the visit of each state. Finally, the states in set S becomes the initial states of the reversed automaton M_R and state q_0 becomes its only final state.

The basic properties of the maximum length automaton and the algorithm for its construction are summarized in the following lemma.

Lemma 16. *Given an acyclic deterministic finite automaton M' with n transitions, Algorithm* MaxLen-Atm *correctly constructs a deterministic finite automaton M accepting language $\mathcal{L}(M) = MaxLen(\mathcal{L}(M'))$ in $\mathcal{O}(n)$ time. M has at most as many states and at most as many transitions as M'.*

4.2.2. Minimum length automaton

The Algorithm MinLen-Atm for the construction of minimum length automaton can be obtained from Algorithm MaxLen-Atm by replacing the Step 1 by the following step:

(i) Transform automaton M into automaton M_R such that $\mathcal{L}(M_R) = MinLen(\mathcal{L}(M))^R$, using the shortest-path algorithm based on the breadth-first-search, considering the transitions of M to be unit-cost edges.

In case of the *DSCS* problem, the input of the stage is a deterministic automaton M_S accepting the set $\mathcal{L}(M_S) = \bigcap_{j=1}^{k} Super(\tilde{x}_j)$. Here, we employ, as the first step of the algorithm, a modification of the shortest-path algorithm for graphs with unit-cost edges based on the breadth-first-search algorithm working in linear time with respect to the number of edges of the input graph [7]. This works, because, a transition diagram of a finite automaton can be seen as a graph with unit-cost edges. The formal steps are described in Algorithm 5.

For each state q of the automaton M, the algorithm keeps the current length $len[q]$ of the shortest path from the initial state q_0. Each state can have one of three possible status', namely *FRESH* (not yet visited), *OPEN* (visited but child states are not yet processed) and *CLOSE* (all child states are processed). The algorithm keeps track of the status in $status[q]$. The algorithm uses queue C for *OPEN* states.

At first, the initial state q_0 of the automaton M, is put into C. Each time a state p is dequeued, each of its child state q is visited. If $status[q]$ is *FRESH*, then its $len[q]$ is set to $len[p] + 1$ and p is set as its predecessor on the shortest path from the initial state. If state q is already in the queue (i.e. *OPEN*) and the length of the shortest path from q_0 via p is equal to $len[q]$, then state p is set as its another predecessor on the shortest path from the initial state. Predecessors are memorized in the form of reversed transition. That is why the resulting automaton of this stage accepts reversed shortest strings of L. The algorithm keeps a set S of final states being targets of the shortest paths form state q_0 in the graph. This set is realized as the states are removed from the queue C. Finally, the states in set S becomes the initial states of the reversed automaton M_R and state q_0 becomes its the only final state.

Notably, the input automaton is not required to by acyclic in this case. The basic properties of the minimum length automaton and of the algorithm for its construction are summarized in the following lemma.

```
procedure: Rev-MaxLen-Atm
input: M = (Q, Σ, δ, q0, F) - acyclic DFA
output: M_R - NFA, L(M_R) = MaxLen(L(M))^R
 1 begin
 2                                                                  initialization
 3   for all q ∈ Q do
 4     d_in[q] ← the number of transition leading to q                input degree
 5     len[q] ← −∞                              length of current longest path from q0
 6   end for
 7   len[q0] ← 0
 8   S ← {q0}                       states with so far maximum length longest path from q0
 9   len_S ← 0                          length of longest path from q0 to states in S
10                                                                  M_R construction
11   C ← New-Queue()                              queue for closed states, d_in[q] = 0
12   Enqueue(C, q0)
13   while not Empty(C) do
14     p ← Dequeue(C)
15     for all s ∈ Σ do
16       q ← δ(p, s)
17       d_in[q] ← d_in[q] − 1
18       if d_in[q] = 0 then Enqueue(C, q)                                 q is closed
19       if len[q] < len[p] + 1 then
20         len[q] ← len[p] + 1                               new longest path from q0
21         δ_R(q, s) ← {p}                                   set the only predecessor
22       else if len[q] = len[p] + 1 then
23                                                       another longest path from q0
24         δ_R(q, s) ← δ_R(q, s) ∪ {p}                         add another predecessor
25       end if
26       if q ∈ F then
27         if len_S < len[q] then
28           len_S ← len[q]                         new longest path from q0 in graph
29           S ← {q}                                    memorize the only target node
30         else if len_S = len[q] then
31                                                  another longest path from q0 in graph
32           S ← S ∪ {q}                                      add another target node
33         end if
34       end if
35     end for
36   end while
37   M_R ← (Q, Σ, δ_R, S, {q0})
38   return M_R
39 end
```

Algorithm 4. Construction of nondeterministic finite automaton M_R for deterministic acyclic finite automaton M accepting language $\mathcal{L}(M_R) = MaxLen(\mathcal{L}(M))^R$.

Lemma 17. *Given a deterministic finite automaton M', with the number of transitions n, Algorithm* MinLen-Atm *correctly constructs finite automaton M, accepting language $\mathcal{L}(M) = MinLen(\mathcal{L}(M'))$ in $\mathcal{O}(n)$ time. M has at most as many states and at most as many transitions as automaton M'.*

4.3. Complexity

Algorithm LCSub-Atm (resp. Algorithm SCSuper-Atm is based on Lemma 12 (resp. Lemma 13). Individual correctness of the component algorithms and the fact that the relevant preconditions are met follow from the discussions and lemmas in the corresponding sections. Therefore, the correctness of the algorithms follow. Here, we deduce the running time of the algorithms.

Lemma 18. *Given a set of $k \geqslant 2$ degenerate strings $S = \{\tilde{x}_1, \tilde{x}_2, \ldots, \tilde{x}_k\}$, where $\tilde{x}_j = \tilde{x}_{j1}\tilde{x}_{j2}\ldots\tilde{x}_{jn_j}$ for $2 \leqslant j \leqslant k$, Algorithm* LCSub-Atm *correctly constructs finite automaton M accepting language $\mathcal{L}(M) = LCSub(S)$ in $\mathcal{O}(|\Sigma|\prod_{j=1}^{k} n_j)$ time.*

Proof. Each automaton $M_j, 1 \leqslant j \leqslant k$, is constructed in $\mathcal{O}(|\Sigma| n_j)$ time (Lemma 10) and it has exactly $n_j + 1$ states (Lemma 10). Therefore, automaton M' is constructed in $\mathcal{O}(|\Sigma|\prod_{j=1}^{k} n_j)$ time (Lemma 15) and automaton M' has at most $\prod_{j=1}^{k}(n_j + 1)$ states. Hence, construction of M from M' takes $\mathcal{O}(|\Sigma|\prod_{j=1}^{k} n_j)$ time (Lemma 16). So, the total time complexity of the algorithm is $\mathcal{O}(|\Sigma|\prod_{j=1}^{k} n_j)$. $\square$

```
procedure: Rev-MinLen-Atm
input: M = (Q, Σ, δ, q0, F) - DFA
output: M_R - NFA, L(M_R) = MinLen(L(M))_R

 1 begin
 2                                                      initialization
 3   for all q ∈ Q do
 4     status[q] ← FRESH                                status FRESH/OPEN/CLOSED
 5     len[q] ← +∞                                      length of current shortest path from q0
 6   end for
 7   len[q0] ← 0
 8   S ← {q0}                                           states with so far minimum length shortest path from q0
 9   len_S ← +∞                                         length of shortest path from q0 to states in S
10                                                      M_R construction
11   status[q0] ← OPEN
12   C ← New-Queue()                                    queue for OPEN states
13   Enqueue(C, q0)
14   while not Empty(C) do
15     p ← Dequeue(C)
16     if p ∈ F then
17       if len_S > len[p] then
18         len_S ← len[p]                               new shortest path from q0 in graph
19         S ← {p}                                      memorize the only target node
20       else if len_S = len[p] then
21                                                      another shortest path from q0 in graph
22         S ← S ∪ {p}                                  add another target node
23       end if
24     end if
25     for all s ∈ {t | t ∈ Σ ∧ ∃q ∈ Q : δ(p, s) = q} do
26       q ← δ(p, s)
27       if status[q] = FRESH then
28         status[q] ← OPEN
29         len[q] ← len[p] + 1
30         Enqueue(C, q)
31         δ_R(q, s) ← {p}                              set the only predecessor
32       else if status[q] = OPEN and len[q] = len[p] + 1 then
33         δ_R(q, s) ← δ_R(q, s) ∪ {p}                  add another predecessor
34       end if
35     end for
36   end while
37   M_R ← (Q, Σ, δ_R, S, {q0})
38   return M_R
39 end
```

Algorithm 5. Construction of deterministic finite automaton M_M for deterministic acyclic finite automaton M accepting language $\mathcal{L}(M_M) = \mathit{MinLen}(\mathcal{L}(M))$.

Lemma 19. *Given a set of $k \geqslant 2$ degenerate strings $S = \{\tilde{x}_1, \tilde{x}_2, \ldots, \tilde{x}_k\}$, where $\tilde{x}_j = \tilde{x}_{j1}\tilde{x}_{j2}\ldots\tilde{x}_{jn_j}$ for $2 \leqslant j \leqslant k$, Algorithm* SCSuper-Atm *correctly constructs finite automaton M accepting language $\mathcal{L}(M) = \mathit{SCSuper}(S)$ in $\mathcal{O}(|\Sigma|\prod_{j=1}^{k} n_j)$ time.*

Proof. Each automaton M_j, $1 \leqslant j \leqslant k$, is constructed in $\mathcal{O}(|\Sigma|n_j)$ time (Lemma 11) and it has exactly $n_j + 1$ states and $|\Sigma|(n_j + 1)$ transitions (Lemma 11). Therefore, automaton M' is constructed in $\mathcal{O}(|\Sigma|\prod_{j=1}^{k} n_j)$ time (Lemma 15) and automaton M' has at most $\prod_{j=1}^{k}(n_j + 1)$ states and at most $|\Sigma|\prod_{j=1}^{k}(n_j + 1)$ transitions (Lemma 15). Construction of M from M' takes $\mathcal{O}(|\Sigma|\prod_{j=1}^{k} n_j)$ time (Lemma 17). Therefore, the total time complexity of the algorithm is $\mathcal{O}(|\Sigma|\prod_{j=1}^{k} n_j)$. □

The following lemma presents the correctness and time complexity of Algorithm CLCSub-Atm (to solve DCLCS problem).

Lemma 20. *Given a set of $k \geqslant 2$ degenerate strings $S = \{\tilde{x}_1, \tilde{x}_2, \ldots, \tilde{x}_k\}$, where $\tilde{x}_j = \tilde{x}_{j1}\tilde{x}_{j2}\ldots\tilde{x}_{jn_j}$ for $2 \leqslant j \leqslant k$, and degenerate string $\tilde{z}$ of length r, Algorithm* CLCSub-Atm *correctly constructs finite automaton M accepting language $\mathcal{L}(M) = \mathit{CLCSub}(S, \tilde{z})$ in $\mathcal{O}(|\Sigma|(\prod_{j=1}^{k} n_j)r)$ time.*

Proof.

Correctness: The algorithm is based on Lemma 14. Individual correctness of the component algorithms in Algorithm CLCSub-Atm follows from the discussions and lemmas in the corresponding sections. Therefore, it remains to show that the preconditions of those Lemmas are guaranteed. According to Lemma 10, the resulting automata $M_1, M_2, \ldots, M_k$ are acyclic. Therefore, it follows from Lemma 15 that the automaton M' is acyclic too. Since the

automaton M' is acyclic the automaton M''' is acyclic as well (Lemma 15). Finally, since automaton M''' is acyclic, automaton M is acyclic.

Time complexity: Each automaton $M_j, 1 \leqslant j \leqslant k$, is constructed in $\mathcal{O}(|\Sigma|n_j)$ time (Lemma 10) and it has exactly $n_j + 1$ states (Lemma 10). Therefore, automaton M' is constructed in $\mathcal{O}(|\Sigma| \prod_{j=1}^{k} n_j)$ time (Lemma 15) and automaton M' has at most $\prod_{j=1}^{k}(n_j + 1)$ states. Automaton M'' is constructed in $\mathcal{O}(|\Sigma|r)$ time (Lemma 11) and it has exactly $r + 1$ states and $|\Sigma|(r + 1)$ transitions (Lemma 11). Therefore, automaton M''' is constructed in $\mathcal{O}(|\Sigma|(\prod_{j=1}^{k} n_j)r)$ time (Lemma 15) and automaton M''' has at most $(\prod_{j=1}^{k}(n_j + 1))(r + 1)$ states and at most $|\Sigma|(\prod_{j=1}^{k}(n_j + 1))(r + 1)$ transitions (Lemma 15). Hence, construction of M from M''' takes $\mathcal{O}(|\Sigma|(\prod_{j=1}^{k} n_j)r)$ time (Lemma 16). So, the total time complexity of the algorithm is $\mathcal{O}(|\Sigma|(\prod_{j=1}^{k} n_j)(r))$. □

5. Conclusion

In this paper, we have presented efficient algorithms to construct two novel types of finite automata on degenerate strings, namely the *subsequence automaton (SubA)* and the *supersequence automaton* and have shown how they can be used to efficiently solve the LCS, SCS and CLCS problems on degenerate strings. In particular, we have presented linear time algorithms for the construction of the above two automata, namely *SubAtm* and *SuperAtm* and have used them to solve DLCS, DSCS and DCLCS problems. For DLCS and DSCS problems, the worst case running time of the corresponding algorithm is $\mathcal{O}(|\Sigma| \prod_{j=1}^{k} n_j)$ and for DCLCS problem, it is $\mathcal{O}(|\Sigma|(\prod_{j=1}^{k} n_j)r)$. While using the automata approach to solve LCS and SCS problems is not new, to the best of our knowledge, this is the first attempt to handle CLCS problem and problems with degenerate strings using finite automata. We believe that, the subsequence and supersequence automata, along with the techniques used in this paper, can be employed to different other problems involving degenerate strings.

References

[1] A.N. Arslan, Ö. Eğecioğlu, Algorithms for the constrained longest common subsequence problems, Int. J. Found. Comput. Sci. 16 (6) (2005) 1099–1109.
[2] A.N. Arslan, Ö. Eğecioğlu, Algorithms for the constrained longest common subsequence problems, in: Proceedings of the Prague Stringology Club Workshop '04, Prague, 2004, pp. 24–32.
[3] R.A. Baeza-Yates, Searching subsequences, Theoretical Computer Science 78 (1991) 363–376.
[4] A.M. Basin, V.N. Balabolin, V.N. Kryukov, An interactive system for multilevel design of technological processes of flexible production, Vestn. Mashinostr. 2 (1987) 42–44.
[5] L. Bergroth, H. Hakonen, T. Raita, A survey of longest common subsequence algorithms, in: Proceedings of the Symposium on String Processing and Information Retrieval, 2000, A Coru, Spain, IEEE Computer Society Press, 2000, pp. 39–48.
[6] F.Y.L. Chin, A. De Santis, A.L. Ferrara, N.L. Ho, S.K. Kim, A simple algorithm for the constrained sequence problems, Inf. Process. Lett. 90 (4) (2004) 175–179.
[7] T.H. Cormen, C.E. Leiserson, R.L. Rivest, Introduction to Algorithms, MIT Press, 1990.
[8] D. Gusfield, Algorithms on Strings, Trees, and Sequences – Computer Science and Computational Biology, University Press, Cambridge, 1997.
[9] J.J. Hébrard, M. Crochemore, Calcul de la distance par les sous-mots, RAIRO Inform. Théor. Appl. 20 (1986) 441–456.
[10] J.E. Hopcroft, J.D. Ullman, Introduction to Automata, Languages and Computations, Addison-Wesley, Reading, MA, 1979.
[11] C.S. Iliopoulos, M.S. Rahman, New efficient algorithms for the LCS and constrained LCS problems, Inf. Process. Lett. 106 (1) (2008) 13–18.
[12] C.S. Iliopoulos, M.S. Rahman, W. Rytter, Algorithms for two versions of lcs problem for indeterminate strings, in: International Workshop on Combinatorial Algorithms (IWOCA), 2007, pp. 93–106.
[13] Y.S. Itoga, The string merging problem, BIT 21 (1) (1981) 20–30.
[14] T. Jiang, M. Li, On the approximation of shortest common supersequences and longest common subsequences, SIAM J. Comput. 24 (5) (1995) 1122–1139.
[15] N.M. Kapustin, Development of Technological Parts Machining Processes by Computer, Mashinostroenie, Moscow, 1976 (in Russian).
[16] R. Lowrance, R.W. Wagner, An extension to the string-to-string correction problem, J. ACM 22 (2) (1975) 177–183.
[17] S.Y. Lu, K.S. Fu, A sentence-to-sentence clustering procedure for pattern analysis, IEEE Trans. Syst. Man. Cybern. SMC-8 (1978) 381–389.
[18] D. Maier, The complexity of some problems on subsequences and supersequences, J. Assoc. Comput. Mach. 25 (2) (1978) 322–336.
[19] B. Melichar, J. Holub, P. Mužátko, Languages and Translation, Publishing House of CTU, 1997.
[20] M. Rodeh, V.R. Pratt, S. Even, Linear algorithm for data compression via string matching, J. ACM 28 (1) (1981) 16–27.
[21] D. Sankoff, R.J. Cedergren, A test for nucleotide sequence homology, J. Molec. Biol. 77 (1973) 159–164.
[22] K. Tempelhof, H. Lichtenberg, Technological standardization as a prerequisite for computer-aided design of technological processes, in: N.M. Kapustin (Ed.), Computer-Aided Design of Technological Processes, Mashinostroenie, Moscow, 1976, pp. 109–153 (in Russian).
[23] V.G. Timkovsky, The complexity of subsequences, supersequences and related problems, Kybern. 5 (1989) 1–13.
[24] Z. Troníček, B. Melichar, Directed acyclic subsequence graph, in: Proceedings of the Prague Stringology Club Workshop '98, Prague, 1998 pp. 107–118.
[25] Y.T. Tsai, The constrained common subsequence problem, Inf. Process. Lett. 88 (2003) 173–176.
[26] R.A. Wagner, M.J. Fischer, The string-to-string correction problem, J. ACM 21 (1) (1974) 168–173.