FOUNDATIONS OF COMPUTING AND DECISION SCIENCES
Vol. 41 (2016) No. 4
ISSN 0867-6356
e-ISSN 2300-3405
DE GRUYTER OPEN
DOI: 10.1515/fcds-2016-0016

INDUCTIVE SYNTHESIS OF COVER-GRAMMARS WITH THE HELP OF ANT COLONY OPTIMIZATION

Wojciech WIECZOREK [*]

Abstract. A cover-grammar of a finite language is a context-free grammar that accepts all words in the language and possibly other words that are longer than any word in the language. In this paper, we describe an efficient algorithm aided by Ant Colony System that, for a given finite language, synthesizes (constructs) a small cover-grammar of the language. We also check its ability to solve a grammatical inference task through the series of experiments.

Keywords: context-free cover-grammar (CFCG), grammatical inference, Ant Colony Optimization (ACO), cliques.

1 Introduction

This paper addresses a cover-grammar problem; namely, given a finite alphabet Σ and a finite subset $X \subset \Sigma^+$, find a compact (the smaller the better) context-free grammar G such that, if $L \subset \Sigma^+$ is the language represented by G, then $L \cap \Sigma^{\leq d} = X$, where d is the length of the longest word(s) in the language X. By the size of a grammar we will mean the number of productions (other measures might also be of interest but are not studied here). The problem of obtaining the concise description of words is crucial to the theories of data compression [16], syntactic pattern recognition [2], and grammatical inference [14]. As far as applications are concerned, it can be applied to the following areas [13]: robotics and control systems, computational linguistics, speech recognition, automatic translation, molecular biology, time series prediction, and data mining.

The concept of covering a finite language is not new. A similar concept has been studied in the context of automata [3]. Although any n-state deterministic finite automaton for some finite language L can be converted into a corresponding minimal

[*] Faculty of Computer Science and Materials Science, University of Silesia, Poland, e-mail: wojciech.wieczorek@us.edu.pl

cover-automaton, using only $O(n \log n)$ time [17], the construction of a minimal cover-grammar seems to be intractable, specially in view of the following facts: (1) there is no polynomial-time algorithm for obtaining the smallest context-free grammar that generates exactly one given word (unless P = NP) [4]; (2) context-free grammar equivalence and even equivalence between a context-free grammar and a regular expression are undecidable [15]; (3) for any alphabet of a size of at least 2, the class of context-free grammars is not polynomially characterizable [12]; (4) the grammar can be exponentially smaller than any word in the language (an example is given in a book [14]). To our best knowledge there are no published algorithms for a cover-grammar problem defined as above. There is a work by Chirathamjaree and Ackroyd [5] on the inference of non-recursive context-free grammars, but their algorithm generates grammars that produce only a given set of strings (has no ability to generalize). So we decided to compare our algorithm with a selected grammatical inference algorithm [21].

The paper's content is organized into seven sections. In Section 2 we present necessary definitions and facts originating from graph theory, combinatorics on words, formal languages and a swarm intelligence method called the ant colony optimization algorithm. Section 3 gives theoretical bases from which the algorithm is developed. The computationally hardest part of the induction algorithm is to repeatedly find a large clique in a graph. The way that this is done by means of ACO (the Ant Colony Optimization) is described in Section 4. Section 5 presents the proposed procedure of the construction of CFCGs. Section 6 shows the experimental results of our approach. Concluding comments are contained in Section 7.

2 Preliminaries

In this section, we are going to describe some definitions and facts about graphs, grammars, and meta-heuristics in order to make the notation understandable to the reader. For further details about the definitions, the reader is referred to the books by Bondy and Murty [1] (graphs), Hopcroft et al. [15] and Lothaire [18] (words and languages), Du and Ko [9] (context-free grammars), and Dréo et al. [8] (meta-heuristics).

2.1 Graphs

A *graph* G is a finite non-empty set of objects called *vertices* together with a (possibly empty) set of unordered pairs of distinct vertices of G called *edges*. The vertex set of G is denoted by $V(G)$, while the edge set is denoted by $E(G)$. The edge $e = \{u, v\}$ is said to *join* the vertices u and v. If $e = \{u, v\}$ is an edge of a graph G, then u and v are *adjacent vertices*, while u and e are *incident*, as are v and e. Furthermore, if e_1 and e_2 are distinct edges of G incident with a common vertex, then e_1 and e_2 are *adjacent edges*. If $v \in V(G)$, then the set of vertices adjacent to v in G is denoted by $N(v)$. The number $|N(v)|$, denoted by $d_G(v)$, is called the *degree* of a vertex v in a graph G.

Given a graph G, there is a natural way of deriving smaller graphs from G. If v

is a vertex of G, we may obtain a graph on $n-1$ vertices by deleting from G the vertex v together with all the edges incident with v. The resulting graph is denoted by $G-v$, and is an example of a subgraph of G. More generally, a graph F is called a *subgraph* of a graph G if $V(F) \subseteq V(G)$, $E(F) \subseteq E(G)$. If D is the set of vertices deleted, the resulting subgraph is denoted by $G-D$. Sometimes, the focus of interest is the set $Y = V(G) - D$ of vertices which remain. In such cases, the subgraph is denoted by $G[Y]$ and referred to as the subgraph of G *induced by* Y. Thus $G[Y]$ is the subgraph of G whose vertex set is Y and whose edge set consists of all edges of G which have both ends in Y.

In a graph G, a *clique* is a subset of the vertex set $C \subseteq V(G)$ such that every two vertices in C are adjacent. If a clique does not exist exclusively within the vertex set of a larger clique then it is called a *maximal clique.* A *maximum clique* is a clique of the largest possible size in a given graph.

2.2 Words and languages

An *alphabet* is a finite, non-empty set of symbols. We use the symbol Σ for an alphabet. A *word* (or sometimes *string*) is a finite sequence of symbols chosen from an alphabet. For a word w, we denote by $|w|$ the length of w. The *empty word* ϵ is the word with zero occurrences of symbols. Let x and y be words. Then xy denotes the *catenation* of x and y, that is, the word formed by making a copy of x and following it by a copy of y. We denote as usual by Σ^* the set of all words over Σ and by Σ^+ the set $\Sigma^* - \{\epsilon\}$. A word w is called a *prefix* (resp. a *suffix*) of a word u if there is a word x such that $u = wx$ (resp. $u = xw$). The prefix or suffix is *proper* if $x \neq \epsilon$. Let $X, Y \subset \Sigma^*$. The *catenation* (or product) of X and Y is the set $XY = \{xy \mid x \in X,\, y \in Y\}$. In particular, we define

$$X^0 = \{\epsilon\}, \quad X^{n+1} = X^n X \ (n \geq 0), \quad X^{\leq n} = \bigcup_{i=0}^{n} X^i.$$

For $w \in \Sigma^*$, we define the *left quotients*

$$w^{-1}X = \{u \in \Sigma^* \mid wu \in X\}.$$

A set of words all of which are chosen from some Σ^*, where Σ is a particular alphabet, is called a *language.* A language is said to possess a *decomposition* [19,25,26] if it can be written as a catenation of two languages neither one of which is the singleton language consisting of the empty word. Languages which are not such products are called *primes.* Thus, having given a finite decomposable (not prime) language L, we can determine both factors—such languages L_1, L_2 that $L = L_1L_2$, $L_1 \neq \{\epsilon\}$ and $L_2 \neq \{\epsilon\}$.

2.3 Context-free grammars

It is very convenient to define languages by grammars. A *context-free grammar* (CFG) is defined by a quadruple $G = (V, \Sigma, P, S)$, where V is an alphabet of *variables* (or sometimes *non-terminal symbols*), Σ is an alphabet of *terminal symbols* such that $V \cap \Sigma = \emptyset$, P is a finite set of *production rules* of the form $A \to \alpha$ for $A \in V$ and $\alpha \in (V \cup \Sigma)^*$, and S is a special non-terminal symbol called the *start symbol.* For the sake of simplicity, we will write $A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_k$ instead of $A \to \alpha_1$, $A \to \alpha_1, \ldots, A \to \alpha_k$. We call a word $x \in (V \cup \Sigma)^*$ a *sentential form.* Let u, v be two words in $(V \cup \Sigma)^*$ and $A \in V$. Then, we write $uAv \Rightarrow uxv$, if $A \to x$ is a rule in P. That is, we can substitute word x for symbol A in a sentential form if $A \to x$ is a rule in P. We call this rewriting a *derivation.* For any two sentential forms x and y, we write $x \Rightarrow^* y$, if there exists a sequence $x = x_0, x_1, x_2, \ldots, x_n = y$ of sentential forms such that $x_i \Rightarrow x_{i+1}$ for all $i = 0, 1, \ldots, n-1$. The language $L(G)$ generated by G is the set of all words over Σ that are generated by G; that is, $L(G) = \{x \in \Sigma^* \mid S \Rightarrow^* x\}$. A grammar is said to be *admissible* if for every $X \in V \cup \Sigma$ there exists some $u, v, w \in \Sigma^*$ such that $S \Rightarrow^* uXv$ and $uXv \Rightarrow^* w$. Usually grammar means admissible grammar. A language is called a *context-free language* if it is generated by a context-free grammar. Assume that G is the unknown (target) CFG to be identified. An *example* (a *positive word*) of G is a word in $L(G)$ and a *counterexample* (a *negative word*) of G is a word not in $L(G)$.

We write $G_1 \equiv G_2$ if $L(G_1) = L(G_2)$. The definition of a context-free grammar imposes no restriction whatsoever on the right side of a production. A *normal form* for context-free grammars is one that, although restricted, is broad enough so that any grammar has an equivalent normal form version. Amongst all normal forms for context-free grammars, the most useful and the most well-known ones are the Chomsky normal form (CNF) and the Greibach normal form (GNF). Nevertheless, we will use the less-known, generalized version of the GNF. A grammar is said to be in (ℓ, k)*-normal form* if each of its rules is in one of two possible forms:

(a) $X \to x$, $x \in \Sigma^+$, $X \in V$, $|x| \leq \ell$, or

(b) $X \to xy$, $x \in \Sigma^+$, $y \in V^+$, $X \in V$, $|x| = \ell$, $|y| \leq k$.

Wood [27] has proved that if $L(G_1)$ is non-empty context-free language without ϵ, then there exist a grammar G_2 such that G_2 is in $(\ell, 2)$-normal form and $G_1 \equiv G_2$. For $\ell = 1$ and $k = 2$ a context-free grammar is said to be in two-standard form.

Assume X to be some fixed finite language. By d we denote the maximal length of a word in X. A CFG G is a *context-free cover-grammar (CFCG) for X* iff $L(G) \cap \Sigma^{\leq d} = X$. (We then also say G covers X.) The grammar G in a certain normal form is called minimal if no CFCG for X in the same normal form has fewer rules than G.

2.4 Ant colony optimization

Although one of the earliest areas for which an ant colony algorithm was implemented was the travelling salesman problem (TSP) [6], a significantly large collection of literature is now available on almost all kinds of combinatorial optimization problems: graph colouring, generalized assignment, multidimensional knapsack, constraint satisfaction, sequential scheduling etc. [8].

The idea of the algorithm can be presented in the following general way. Let $U = \{u_1, u_2, \ldots, u_n\}$ be a universum. Suppose that some subsets of U are (maybe incomplete) solutions to a given problem. Let $S \subseteq U$ be such a subset. We will consider a set $J \subseteq U - S$ of elements which can be added to S in order to build a more integral solution. The essential part of an ant colony algorithm relies on selecting a $j \in J$. This selection is non-deterministic and is repeated until a complete solution is found. The choice of j is controlled by two parameters, d_j (attractiveness) and τ_j (the amount of pheromone), which help to maintain balance between diversification/intensification. By intensification, we understand the *exploitation* of the information gathered by the system at a given time. On the other hand, diversification is the *exploration* of search space; this aim is achieved by introducing the random perturbation in the system.

The whole process is performed simultaneously for many subsets, starting from different $S_k = \{u_{i_k}\}$ for $k = 1, 2, \ldots, m$. Exact formulas and a pseudo-code for an ant colony algorithm adapted to determine a maximal clique are defined in Section 4. It is worth mentioning, that original formulas (those devised for TSP-like problems) contain additional parameters: α, β, and ρ controlling the relative importance of the intensity and the diversity, and the process of evaporation of the trails of pheromone. Beside discarding those parameters, we stop the algorithm after the first iteration without improvement, instead of setting a priori the number of iterations. All the modifications with respect to the original algorithm were intended to simplify our implementation and to cause that in each subsequent iteration the importance of d_j diminishes, speeding up the convergence of the method. Please note that obtaining a large maximal clique is an intermediate goal and it is unnecessary to get the largest one.

3 Cliques and the multi-decomposition of a finite language

Let Σ be a finite non-empty alphabet, and $m \geq 1$. A *multi-decomposition* of some finite language $X \subset \Sigma^*$ is a set of concatenations whose union is X:

$$X = \bigcup_{i=1}^{m} L_i R_i, \quad L_i, R_i \subset \Sigma^*,\ L_i, R_i \neq \emptyset,\ L_i, R_i \neq \{\epsilon\}.$$

It is easy to see that every non-empty finite language X which satisfies $X \cap (\Sigma \cup \{\epsilon\}) = \emptyset$ has a multi-decomposition, since for a singleton $\{w\}$, $w = w_1 w_2 \cdots w_k$, $k \geq 2$, we can write $\{w\} = \{w_1\}\{w_2 \cdots w_k\}$. Let $X = \{x_1, x_2, \ldots, x_n\}$ $(n \geq 1)$ be a given language with no words of size 0 (the empty word) or 1. All possible word-splittings are denoted by $x_i = u_{ij} w_{ij}$, $i = 1, 2, \ldots, n$, $j = 1, 2, \ldots, |x_i| - 1$. The prefix u_{ij} consists of the j leading symbols of x_i, while the suffix w_{ij} consists of the $|x_i| - j$ trailing symbols of x_i.

This section shows how the multi-decomposition of a finite language $X \subset \Sigma\Sigma^+$ is connected with the cliques of an undirected graph. Consider the graph G with vertex set

$$V(G) = \bigcup_{i=1}^{n} \{(u_{ij}, w_{ij}) \mid j = 1, 2, \ldots, |x_i| - 1\}$$

and with edge set $E(G)$ (E for short) given by:

$$\{(u_{ij}, w_{ij}),\ (u_{kl}, w_{kl})\} \in E \ \Leftrightarrow\ u_{ij}w_{kl} \in X \ \wedge\ u_{kl}w_{ij} \in X.$$

Let $X = \bigcup_{i=1}^{m} L_i R_i$ be a multi-decomposition. One can readily verify that any concatenation $L_i R_i \subseteq X$ is represented by the corresponding clique:

$$\{(u_{t_1}, w_{t_1}), (u_{t_2}, w_{t_2}), \ldots, (u_{t_r}, w_{t_r})\},$$

where $t_i \in \{1, \ldots, n\} \times \{1, \ldots, d-1\}$, $d = \max_{x \in X} |x|$, and

$$L_i = \bigcup_{j=1}^{r} \{u_{t_j}\}, \quad R_i = \bigcup_{j=1}^{r} \{w_{t_j}\}.$$

Example: Let us consider the language $X = \{ab,\ acc,\ bb,\ bcc,\ ccc\}$. A graph built based on X is shown in Figure 1. It has $8 + 11 + 6 + 1$ (its vertexes, its edges, its triangles, and one 4-vertex clique) cliques, but only three maximal cliques:

$$\begin{aligned} C_1 &= \{(b, cc),\ (a, cc),\ (b, b),\ (a, b)\}, \\ C_2 &= \{(ac, c),\ (bc, c),\ (cc, c)\}, \\ C_3 &= \{(b, cc),\ (a, cc),\ (c, cc)\}. \end{aligned}$$

In order to minimize the number of concatenations in a multi-decomposition, it is recommended that only the ones that are represented by large cliques be chosen. Take C_1 and C_2 for instance: $X = L_1R_1 \cup L_2R_2$, where:

$$\begin{aligned} L_1 &= \{a, b\}, & R_1 &= \{b, cc\}, \\ L_2 &= \{ac, bc, cc\}, & R_2 &= \{c\}. \end{aligned}$$

4 The Use of Ant Colony System

It is a well-known fact that finding a maximum clique is an NP-hard optimization problem as is covering the vertexes of a graph by the minimum number of cliques [11]. What is more, the number of maximal cliques possible in a graph G can be as large as $4.3^{[|V(G)|/3]-1}$ [20]. Hence, as regards the minimization of concatenations in a multi-decomposition (which leads in turn to a decrease in the number of production rules), a randomized routine presented in Figure 2 is proposed. As for line 2, the graph is constructed in exactly the same way as described earlier in the previous section. We say that a word $x \in X$ is covered if it has been added to a set Y (initiated in line 3). It

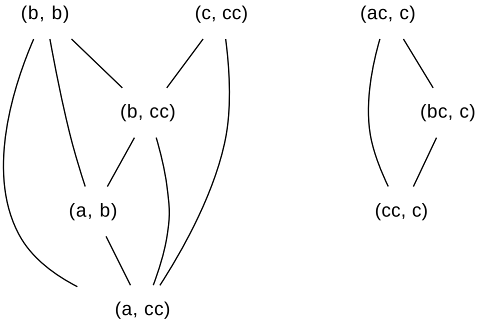

Figure 1: The graph G for $X = \{ab, acc, bb, bcc, ccc\}$.

```
1: procedure Cover($X$)
2:     $G :=$ a graph built based on the language $X$
3:     $Y := \emptyset$                                  ▷ the set of covered words
4:     while $Y \neq X$ do
5:         $H :=$ the copy of $G$
6:         randomly select a vertex $v = (u, w) \in V(H)$ such that $uw \notin Y$
7:         $H := H[N(v) \cup \{v\}]$
8:         if $H$ is not a clique then
9:             $H :=$ mClique($H$)
10:        $LR :=$ a concatenation determined by $H$
11:        for $x \in LR$ do
12:            $Y := Y \cup \{x\}$
13:        report $LR$ as a next concatenation
```

Figure 2: An algorithm for the multi-decomposition process.

means that x will be an element of a certain concatenation in a multi-decomposition. The algorithm starts with all words in X to be uncovered. To be more precise, there is also a relation between words and vertices which is implied from the algorithm: x_i is an uncovered word iff all vertices (u_{ij}, w_{ij}), $j = 1, 2, \dots, |x_i| - 1$, are uncovered. As regards lines 4–13, the following strategy is used in order to obtain few concatenations in a multi-decomposition. It works by selecting (uniformly at random) one uncovered vertex, v, at a time and determining a maximal clique in a subgraph induced by the set of vertices adjacent to v. In consequence, a corresponding concatenation that covers numerous words is regularly generated (line 13). The pair of sets (L, R) (line 10) is resolved according to formulas also given earlier in the previous section.

The computationally hardest step of procedure Cover, i.e., the selection of a large maximal clique (mClique(G)), has been realized following ant colonies approach [8]. The ant colony algorithm has been specially adapted to suit searching for a large maximal clique. In particular, a probability distribution used by the ants has been modified and pheromone evaporation has been disallowed. In each iteration t, each ant k ($k = 1, \ldots, m$) traverses a fragment of the graph and builds a maximal clique $S^k(t)$. For each ant, the choice of the next vertex j to enlarge a clique S depends on:

1. A set S of the already chosen vertexes, which defines the possible choices in each step, when the ant k is going to collect the next vertex:
$$J_S^k = \{v \in V(G) \mid v \notin S \wedge \forall s \in S, (v, s) \in E(G)\}.$$
2. The degree of a vertex j, $d_G(j)$. This static information is used to direct the choice of the ants towards promising vertexes.
3. Quantity of pheromone deposited in a vertex, $\tau_j(t)$. This can be viewed as a global memory of the system, which evolves through a searching process.

The rule of vertex preference can be stated as following:

$$p_{Sj}^k(t) = \begin{cases} \frac{d(j)+\tau_j(t)}{\sum_{l \in J_S^k}(d(l)+\tau_l(t))}, & j \in J_S^k, \\ 0, & j \notin J_S^k. \end{cases} \tag{1}$$

After a full run, each ant leaves a certain quantity of pheromones $\Delta\tau_j^k(t)$ on its entire clique, the amount of which depends on the size of the solution found:

$$\Delta\tau_j^k(t) = \begin{cases} |S^k(t)| & \text{if } j \in S^k(t), \\ 0 & \text{if } j \notin S^k(t). \end{cases} \tag{2}$$

The process of evaporation of the trails of pheromone has not been put in. Hence, the update rule for the trails is simply given as:

$$\tau_j(t+1) = \tau_j(t) + \sum_{k=1}^{m} \Delta\tau_j^k(t), \tag{3}$$

where m is the number of ants (in the present implementation $m = |V(G)|$ was chosen). The initial quantity of pheromone in the vertexes is zero. Figure 3 exhibits the pseudo-code of the ant colony approach to solve the clique problem.

5 An algorithm for the cover-grammar problem

Suppose we want to find a cover-grammar for a finite language $X \subset \Sigma^+$. In this section we discover a two-phase algorithm that does it efficiently. The idea behind the algorithm is as follows. The language X and possibly other longer words can be generated by a context-free grammar $G = (V, \Sigma, P, S)$ in $(\ell, 2)$-normal form. For some

```
function mClique($G$)
    $t := 0$; $C := \emptyset$; $M := \emptyset$; $S := \emptyset$
    repeat
        $t := t + 1$; $C := M$; $M := \emptyset$
        for $k := 1$ to $m$ do
            $i :=$ a randomly chosen vertex from $V(G)$
            $S := \{i\}$
            while $J_S^k \neq \emptyset$ do
                choose a vertex $j$ from $J_S^k$ according to (1)
                $S := S \cup \{j\}$
            $S^k(t) := S$
            deposit the trails in accordance with (2)
        if $|S| > |M|$ then
            $M := S$
        update trails according to (3)
    until $|C| \geq |M|$
    return $C$
```

Figure 3: The ant colony algorithm for finding a large maximal clique.

ℓ (which is determined by hand) we can write:

$$\begin{aligned}
&S \to t_1 \mid t_2 \mid \ldots \mid t_j \\
&S \to u_1\, B_1 \mid u_1\, C_{11}\, D_{11} \mid u_1\, C_{12}\, D_{12} \mid \ldots \mid u_1\, C_{1m_1}\, D_{1m_1} \\
&S \to u_2\, B_2 \mid u_2\, C_{21}\, D_{21} \mid u_2\, C_{22}\, D_{22} \mid \ldots \mid u_2\, C_{2m_2}\, D_{2m_2} \\
&\vdots \quad \vdots \\
&S \to u_k\, B_k \mid u_k\, C_{k1}\, D_{k1} \mid u_k\, C_{k2}\, D_{k2} \mid \ldots \mid u_k\, C_{km_k}\, D_{km_k}
\end{aligned}$$

where $t_i \in X$, $i = 1, 2, \ldots, j$, are all words in X that are of length $|t_i| \leq \ell$; $u_i \in \Sigma^+$ for $i = 1, 2, \ldots, k$ are all proper prefixes of words in X that are of length $|u_i| = \ell$; $\bigcup_{i=1}^{m_q} C_{qi}\, D_{qi}$ for $q = 1, 2, \ldots, k$ are multi-decompositions of $(u_q^{-1}X) - \Sigma$; and $B_q = (u_q^{-1}X) \cap \Sigma$. Then, the following equation holds:

$$X = \left(\bigcup_{i=1}^{j} \{t_i\} \right) \cup \left(\bigcup_{q=1}^{k} \{u_q\} B_q \right) \cup \left(\bigcup_{q=1}^{k} \bigcup_{i=1}^{m_q} \{u_q\} C_{qi}\, D_{qi} \right).$$

Proceeding in the same way as described above, for every set B, C and D, and then recursively for the next sets, we obtain the grammar that generates exactly the language X. This finishes the first phase. A pseudo-code for obtaining this initial grammar is given in Section 5.1. In the second phase the size of the grammar is reduced by merging some non-terminals. The possibility of getting a grammar that generates an infinite language is the advantageous by-product of this reduction.

```
function Rules($X$)
    $V[X]$ := the consecutive number $i$ from $(0, 1, 2, \ldots)$
    $U := \{p \in \Sigma^{\leq \ell} \mid (p \in X) \vee (pw \in X,\ |p| = \ell)\}$
    for $u \in U$ do
        if $|u| < \ell$ then
            add to $P$ the rule $V_i \to u$
        else
            $A := u^{-1}X$
            if $\epsilon \in A$ then
                add to $P$ the rule $V_i \to u$
                $A := A - \{\epsilon\}$
            $B := A \cap \Sigma$
            if $B \neq \emptyset$ then
                $b := V[B]$ if $B \in V$ else Rules($B$)
                add to $P$ the rule $V_i \to u\, V_b$
            $A := A - B$
            if $A \neq \emptyset$ then
                for $CD \in$ Cover($A$) do
                    $c := V[C]$ if $C \in V$ else Rules($C$)
                    $d := V[D]$ if $D \in V$ else Rules($D$)
                    add to $P$ the rule $V_i \to u\, V_c\, V_d$
    return $i$
```

Figure 4: The method of constructing the initial grammar for a set X.

5.1 Phase 1—constructing an initial grammar

Our method of constructing the initial grammar is based on the idea that was demonstrated at the beginning of this section and is presented as function Rules (Figure 4). Before the execution of Rules(X), the set P and the map V (an associative array where a key is the set of words and a value is an integer index) are empty. To every set L in V is associated a number i ($V[L] = i$) such that a grammar variable V_i 'represents' the set L. Actually, $L = \{w \in \Sigma^+ \mid V_i \Rightarrow^* w\}$. $V[X] = 0$ and, thus, V_0 is the start symbol. No lines of this algorithm are in need of an explanation—the pseudo-code is already written in nearly a high level programming language.

5.2 Phase 2—merging non-terminals

In this subsection, we will describe function Reduce, which—in an iterative process—improves the current grammar by reducing the number of variables and consequently the number of rules. This reduction leads in turn to an increase in the number of accepted words and this is why, in some cases, we get a grammar that generates an infinite language. As can be seen from the pseudo-code of Figure 5, not all pairs

```
 1: function REDUCE($V$, $P$)
 2:     $z := |V|$
 3:     $T$ := an array that fulfills $T[j] = J \Leftrightarrow V[J] = j$
 4:     for $0 \le i < j < z$ do
 5:         if $T[i] \ne \text{nil} \wedge T[j] \ne \text{nil}$ then
 6:             $P' := \emptyset$
 7:             for $(A \to \alpha) \in P$ do
 8:                 $A' \to \alpha'$ := substitute $V_i$ for $V_j$ in $A \to \alpha$
 9:                 $P' := P' \cup \{A' \to \alpha'\}$
10:             $G$ := the grammar determined by $P'$
11:             if $L(G) \cap \Sigma^{\le d} = X$ then
12:                 $T[j] := \text{nil}$
13:                 $P := P'$
14:     return $P$                                  ▷ the reduced set of rules
```

Figure 5: An algorithm for merging non-terminals.

of non-terminals are allowed to be merged. So as to avoid the acceptance of a word $y \notin X$, $|y| \le d$ (d is the length of the longest word in X), after the mergence (in lines 7–9) of non-terminals V_i and V_j, the improvement process is controlled with the help of the enumeration of words accepted by G. It means that for $i = 1, 2, \ldots, d$ the set $L(G) \cap \Sigma^i$ is listed and checked for the equality with $X \cap \Sigma^i$ (line 11).

How can we be sure that the mergence of two variables V_i and V_j $(i < j)$ only increases the number of accepted words? The skeleton of a proof would rely on a claim that if for any word $x \in \Sigma^+$ before the mergence $V_0 \Rightarrow^* u\,V_j\,w \Rightarrow^* x$ $(u, w \in (V \cup \Sigma)^*)$ holds, then after the mergence we could write a like derivation, putting in the place of V_j a variable V_i.

5.3 The enumeration of words

In order to check the condition in line 11 of function REDUCE, we need a routine for generating all words of length i, for $i = 1, 2, \ldots, d$, that are accepted by a given grammar G. Towards this goal, we aim to define a function ζ determining the words of a certain length and to provide a guide for its efficient implementation.

Let $G = (V, \Sigma, P, S)$ be a context-free grammar in $(\ell, 2)$-normal form. Let the production rules P be written in the form:

$$P = \{ \quad \begin{array}{llll} V_0 \to \alpha_{00}, & V_0 \to \alpha_{01}, & \ldots, & V_0 \to \alpha_{0r_0}, \\ V_1 \to \alpha_{10}, & V_1 \to \alpha_{11}, & \ldots, & V_1 \to \alpha_{1r_1}, \\ \multicolumn{4}{l}{\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots} \\ V_z \to \alpha_{z0}, & V_z \to \alpha_{z1}, & \ldots, & V_z \to \alpha_{zr_z} \quad \}, \end{array}$$

where: $\{V_0, V_1, \ldots, V_z\} = V$, $\alpha_{ij} \in (V \cup \Sigma)^+$, $S = V_0$. If $\alpha_{ij} = \gamma_1\gamma_2\cdots\gamma_k$ and $\gamma_m \in V \cup \Sigma$ we will write α_{ijm} for γ_m $(1 \le m \le k)$.

For $\beta \in V \cup \Sigma$ and natural $n \geq 1$ by $\zeta(\beta, n)$ we mean the set of words of length n which can be derived from β in the grammar G. Observe that for any $a \in \Sigma$ we have $\zeta(a, n) = \{a\}$ exactly when $n = 1$ and $\zeta(a, n) = \emptyset$ in case $n > 1$. The set of words which can be derived from a non-terminal A can be computed as the union of sets of words which can be derived from every sentential form α at the right-hand side of $A \to \alpha$. Similarly, $\zeta(V_i, n)$ can be obtained through joining sets of words of length n which can be derived from every α_{ij}, $j = 1, 2, \ldots, r_i$. When we determine the set of words of length n which can be derived from $\alpha_{ij} = u_1 u_2 \cdots u_\ell B$ ($u_m \in \Sigma$, $B \in V$) only the words of length $n - \ell$ which can be derived from B have to be found, whereas when determining the set of words of length n which can be derived from $\alpha_{ij} = u_1 u_2 \cdots u_\ell C\,D$ ($u_m \in \Sigma$, $C, D \in V$) all compositions[1] of $n - \ell$ with two parts have to be taken into account. For example, let $\alpha = aBC$ and $n = 4$. The set of all words of length four derived from α consists of:

- catenation of $\{a\}$ and all words of length one derived from B and all words of length two derived from C,
- catenation of $\{a\}$ and all words of length two derived from B and all words of length one derived from C.

By combining the facts mentioned above, we can obtain the following recurrence relation:

$$\zeta(\beta, n) = \begin{cases} \{\beta\} & : \beta \in \Sigma \ \wedge \ n = 1, \\ \bigcup_{j=1}^{r_i} \bigcup_{\mathcal{C}(\alpha_{ij}, n)} \zeta(\alpha_{ij1}, c_1) \cdots \zeta(\alpha_{ijk}, c_k) & : \beta = V_i \ \wedge \ n \geq 1, \\ \emptyset & : \text{otherwise}, \end{cases}$$

where $k = |\alpha_{ij}|$ and the sequence $c_1, c_2, \ldots, c_k$ belongs to the compositions

$$\mathcal{C}(\alpha, n) = \begin{cases} \text{if } |\alpha| = n \text{ then } \{(1, 1, \ldots, 1)\} \text{ else } \emptyset & : |\alpha| \leq \ell, \\ \text{if } n > \ell \text{ then } \{(1, 1, \ldots, 1, n - \ell)\} \text{ else } \emptyset & : |\alpha| = \ell + 1, \\ \{(1, 1, \ldots, 1, p, q) \mid p + q = n - \ell \ \wedge \ p, q \geq 1\} & : |\alpha| = \ell + 2. \end{cases}$$

The application of dynamic programming[2] results in the effective implementation of the formula $\zeta(\beta, n)$. If $\Sigma = \{a_1, a_2, \ldots, a_t\}$, it suffices to compute ζ for every entry in the table:

$$[(a_1, 1),\ (a_2, 1), \ldots, (a_t, 1),\ (V_0, 1),\ (V_1, 1), \ldots, (V_z, 1),\ (V_0, 2), \ldots, (V_z, d)]$$

in the given order by reusing stored results.

Let s be $\max_{A \in V,\, 1 \leq i \leq d} |\zeta(A, i)|$. In practical situations the sizes of alphabets of terminal and non-terminal symbols are small constants, $|V \cup \Sigma| = c_1$, and the number of production rules for a non-terminal is also a small constant, c_2. There are no more than $c_1 d$ entries in the table. Since catenation of two sets can be done in time $O(ds^2)$ and there are at most $d - 1$ compositions in $\mathcal{C}(\alpha, d)$, the worst-case running time for every entry is $c_2(d-1)O(ds^2)$. Thus, the evaluation of $L(G) \cap \Sigma^{\leq d}$ consumes $c_1 d c_2 (d-1) O(ds^2) = O(d^3 s^2)$ time.

[1] Compositions are merely partitions in which the order of summands is considered. For example, there are four compositions of 3: (3), (12), (21), (111).

[2] To become more familiar with dynamic programming please consult Chapter Eighteen of [23], Chapter Fifteen of [7], or Chapter Nine of [24].

5.4 A run of the algorithm

We run the algorithm on an example. Let the sample language consist of the words:

> $X = \{a,\ aa,\ aab,\ aba,\ baa,\ aaa,\ aaaa,\ aaab,\ aaba,\ abaa,\ baaa,\ aaaaa,\ aaaab,\ aaaba,\ aabaa,\ abaaa,\ baaaa,\ aaabb,\ aabba,\ abbaa,\ bbaaa,\ baaab,\ baaba,\ babaa,\ abaab,\ aabab,\ ababa\}.$

Let us assign $\ell := 1$. At the beginning, a set of production rules P and a map V representing non-terminals are both empty. $V[X] := 0$. There are two sets to consider in an initial call—Rules(X):

> $a^{-1}X = \{\epsilon,\ a,\ ab,\ ba,\ aa,\ aaa,\ aab,\ aba,\ baa,\ aaaa,\ aaab,\ aaba,\ abaa,\ baaa,\ aabb,\ abba,\ bbaa,\ baab,\ abab,\ baba\},$
>
> $b^{-1}X = \{aa,\ aaa,\ aaaa,\ baaa,\ aaab,\ aaba,\ abaa\}.$

First, $a^{-1}X$ is processed. The rule $V_0 \to a$ is added to P; $V[\{a\}] := 1$; the rules $V_1 \to a$ (this rule is added in the second recursive call) and $V_0 \to a\,V_1$ are added to P. After the removal of ϵ and a from $a^{-1}X$, the following concatenations are generated by Cover($a^{-1}X - (\{\epsilon\} \cup \Sigma)$):

$$\begin{aligned} &\{a\}\,\{a, aba, aa, b, aaa, ba, aab, abb, bba, ab, bab, baa\} &&(V_1\,V_2),\\ &\{a, b\}\,\{a, aa, aaa, aab, aba, baa\} &&(V_3\,V_4). \end{aligned}$$

After the return from appropriate recursive calls, the rules $V_0 \to a\,V_1\,V_2$ and $V_0 \to a\,V_3\,V_4$ are added to P.

Similarly, $b^{-1}X$ is processed. The following concatenations are generated by Cover($b^{-1}X$):

$$\begin{aligned} &\{a, aa, aaa, aab, aba, baa\}\,\{a\} &&(V_4\,V_1),\\ &\{a\}\,\{a, aa, aaa, aab, aba, baa\} &&(V_1\,V_4). \end{aligned}$$

After return from appropriate recursive calls, the rules $V_0 \to b\,V_4\,V_1$ and $V_0 \to b\,V_1\,V_4$ are added to P.

The subsequent sets of words, which represent variables V_1, V_2, V_3, and V_4 are processed in recursive calls. Based on the set $\{a\}$ we get $V_1 \to a$. Based on the set $\{a, aba, aa, b, aaa, ba, aab, abb, bba, ab, bab, baa\}$ we get

$$V_2 \to a \mid a\,V_3 \mid a\,V_3\,V_3 \mid b \mid b\,V_1 \mid b\,V_3\,V_1 \mid b\,V_1\,V_3$$

Based on the set $\{a, b\}$ we get

$$V_3 \to a \mid b$$

Finally, based on the set $\{a, aa, aaa, aab, aba, baa\}$ we get

$$V_4 \to a \mid a\,V_1 \mid a\,V_1\,V_3 \mid a\,V_3\,V_1 \mid b\,V_1\,V_1$$

Afterwards, the resultant grammar is 'reduced' by merging some variables. All pairs $(V_0, V_1), (V_0, V_2), \ldots, (V_1, V_2), \ldots, (V_4, V_5)$ are checked. It appears that three pairs of variables can be successfully merged. These are V_0 with V_1, V_0 with V_4, and V_2 with V_3. So a cover-grammar for X is:

$$V_0 \to a \mid a\,V_0 \mid a\,V_0\,V_2 \mid a\,V_2\,V_0 \mid b\,V_0\,V_0$$
$$V_2 \to a \mid b \mid a\,V_2 \mid b\,V_0 \mid a\,V_2\,V_2 \mid b\,V_0\,V_2 \mid b\,V_2\,V_0$$

We have empirically verified—till the length eighteen—that this grammar accepts all and only the words from the language $\{\{a, b\}^+ \mid \text{every word has more } a\text{'s than } b\text{'s}\}$.

5.5 Complexity issues

Let n be the size of an input language X, and d be the length of the longest word in X. It is not hard to see that the running time of the second phase is polynomially bounded, even if the size of an obtained grammar is proportional to $d \times n$. The outermost loop of function Reduce iterates at most $O((dn)^2)$ times. The enumeration of words is its dominant operation. Because c_1 and c_2 from Subsection 5.3 can be as large as $O(dn)$ and $s = O(n)$, in the pessimistic case the second phase takes no more than $O(d^5n^4)$ time.

As far as the first phase is concerned, the running time T of the subroutine Rules can be assessed by the following relation:

$$T(d, n) = \sum_{i=1}^{c_1} \big(O(d^2n^4) + c_2T(d - \ell,\, n/c_3)\big)$$

where c_1 is the number of prefixes (see line 3 of Rules); the $O(d^2n^4)$ component denotes the number of operations in the subroutine Cover and other operations in Rules apart from recursive calls; c_2 is the number of recursive calls in lines 19 and 20 (a recursive call in line 14 does not have to be considered since $|B| \leq |\Sigma|$ and its execution time is already contained in $O(d^2n^4)$); c_3 is a scaling factor such that n/c_3 denotes the size of languages in consecutive recursive calls.

Hypothetically, c_1 and c_2 can be as large as n, and c_3 can be equal to one. Then, for small ℓ, the algorithm would have an exponential time complexity. However, usually c_1 and c_2 are small constants, and $c_3 \geq 2$. Then the recurrence leads to $T(d, n) = O(c^{\min\{d/\ell,\, \log_2 n\}}d^2n^4)$ where c is a small constant. This formula reveals that the value of ℓ should be relatively large if we want to shorten the running time of the whole process. But then it may be unable to achieve minimal CFCGs. Fortunately, for formal languages considered in practice, the execution of the algorithm can be performed very fast even though $\ell = 1$, n is of the order of a few hundreds and d is of the order of a dozen or so.

6 Experimental results

In all experiments we used the implementation[3] of algorithms written in Python. An interpreter ran on an Intel i3-4010U, 1.7 GHz processor under Windows 10 operating system with 16 GB RAM. As regards the creation and manipulation of graphs, we took advantage of the NetworkX[4] library (version 1.11).

The goal of experiments is to show that not only is our algorithm able to generate short cover-grammars for finite languages, but first of all it might identify infinite languages based on their finite examples. The benchmark is composed of seven context-free, not regular languages and single (the third) regular language ($\sharp_x(w)$ denotes the number of xs in the word w):

- $L_1 : a^m b^n,\ 1 \leq m \leq n$,
- $L_2 :$ balanced parentheses,
- $L_3 : a^m b^n,\ m \geq 1$ and $n \geq 1$,
- $L_4 : \{w \mid w \in \{a,b\}^+$ and $\sharp_a(w) = \sharp_b(w)\}$,
- $L_5 : \{w \mid w$ is a palindrome and $w \in \{a,b\}\{a,b\}^+\}$,
- $L_6 : \{w \mid w \in \{a,b\}^+$ and $2\sharp_a(w) = \sharp_b(w)\}$,
- $L_7 :$ the language of Łukasiewicz ($S \to aSS;\ S \to b$),
- $L_8 : \{a^i b^j c^k \mid i = j$ or $j = k,\ i,j,k \geq 1\}$.

The first six languages were considered by Nakamura et al. [21,22], and the language L_7 was considered by Eyraud et al. [10], while the inherently ambiguous context-free language L_8 was the one language for which the Synapse system made by Nakamura and Matsumoto [22] could not directly synthesize a grammar from positive and negative sample words.

For each target language we generated a learning and a test sample in the following way. We built the learning sample by listing the set of words $L \cap \Sigma^{\leq d}$ for each language. The maximum length $d = 7$ ($d = 12$ for L_8) was set up such that we could compare our running times with those obtained by Nakamura and Ishiwata [21]. Regarding the construction of the test set, we generated the set of all words up to the length twelve (and fourteen for L_8) over the alphabet of terminal symbols used to define the target language. The test sequences were then labelled as positive or negative depending on their membership in the language.

In order to study the behaviour of our algorithm, for each target language we fixed $\ell := 1$ and constructed a CFCG by applying the first (creating an initial grammar) and the second phase (reduction) of the algorithm. Then, we evaluated the inferred cover-grammar on the test set by checking if it correctly classifies all test sequences. Such a procedure has been applied 30 times and the results are gathered in Table 1. For each

[3] https://github.com/wieczorekw/wieczorekw.github.io

[4] http://networkx.lanl.gov/

	t			$\lvert P\rvert$			$\lvert V\rvert$						
L	Min	Max	Ave	Min	Max	Ave	Min	Max	Ave	CS	t	$\lvert P\rvert$	$\lvert V\rvert$
1	0.01	0.25	0.14	9	19	13.7	5	8	6.8	23	1	4	2
2	0.01	0.03	0.01	9	20	13.3	5	10	7.6	16	1	4	2
3	0.04	1.53	0.62	12	21	18.3	8	12	9.8	30	6	6	2
4	0.03	0.08	0.05	17	29	22.4	8	12	10.3	26	2	7	2
5	0.02	1.37	0.57	12	73	47.2	6	31	18.6	11	2	10	3
6	0.02	0.08	0.04	16	27	20.5	8	12	9.5	14	22	9	3
7	0.01	0.07	0.03	12	26	17.6	6	13	9.0	28	not considered		
8	14.61	153.72	40.96	62	303	163	36	107	65.6	5	unable to infer		

Table 1: Inferred grammars' characteristics and CPU time t of computations (sec.). $|P|$ and $|V|$ denote, respectively, the number of production rules and the number of variables in a generated grammar. The leading columns show our results, while the last three columns concern the results obtained by Nakamura et al. [21, 22].

language correct classification has been reached within CS (Completed Successfully) runs out of 30. The sizes of grammars for L_i, $i = 1, 2, \ldots, 7$, varied from 9 to 73, and the CPU time varied from 0.01 to 1.53 seconds. As for L_8, correct classification has been reached for $d = 12$ only within five runs. In this case computations lasted on average 40.96 seconds and larger grammars have been obtained. The last three columns show results reported by Nakamura et al. [21, 22]. They used Windows version Visual C++ compiler and Intel Pentium II processor with 400 MHz clock. But even if we divide their t by four ($4 \times 0.4 \approx 1.7$) the values will be greater than our minimum t. As can be seen from the table, Nakamura et al. obtained smaller grammars for languages from 1 to 6, but there are three main differences between their approach and ours. For one thing, Synapse needs both positive and negative words. For another, Synapse could not directly synthesize a grammar for language L_8 in a reasonable amount of time. Finally, a current restriction of Synapse is that it has not synthesized grammars with more than about twelve rules.

We owe an explanation of the reasons and principles of comparison between our method and the selected grammatical inference method. The main reason is the lack of other methods for generating CFCGs. Besides, we wanted to emphasize that our method is able to generalize, i.e., giving expected recursive grammars. We should be aware of the differences between these two tasks. Inductive synthesis (IS) of cover-grammars needs all positive words (examples) of an unknown language up to a certain length d. A grammatical inference (GI) method usually takes as input some positive words and some negative words (counterexamples). After all, it could be said that in IS counterexamples are given indirectly: one can easily render all the negative words of the length up to d (absence in examples implies presence in counterexamples), therefore in the set of positive words the information about the negative words is already encoded. So both tasks, IS and GI, have the same goal: to get a grammar which accepts all examples, but no counterexample. However, the possibility of omitting counterexamples gives IS an advantage over GI. It is especially seen when an expected language has a few examples and many counterexamples. Then, IS takes a small input sample, while GI needs much bigger input sample so as to enhance the chance of getting the proper grammar. Let us consider the language $L = \{ab(bbaa)^n bba(ba)^n \mid n \geq 0\}$, which can be generated by the grammar $G = (\{S, A, B, C, D\}, \{a, b\}, P, S)$ with productions P in $(2, 2)$-normal form:

$$\begin{array}{ll} S \to a\,b\,D & A \to a \\ B \to b & C \to a\,a\,D\,B \\ D \to b\,b\,A \mid b\,b\,C\,A & \end{array}$$

The equivalent grammar:

$$\begin{array}{ll} V_0 \to a\,b\,V_1\,V_6 & V_1 \to b \mid b\,b\,V_2\,V_6 \\ V_2 \to a\,a\,V_1 \mid a\,a\,V_4\,V_6 & V_4 \to b\,b\,V_5\,V_1 \\ V_5 \to a\,a & V_6 \to b\,a \end{array}$$

has been achieved with

$$X = L \cap \Sigma^{\leq 17} = \{abbba,\ abbbaabbaba,\ abbbaabbaabbababa\}$$

using less than 0.1 seconds. Please note that in this example the number of negative words is relatively large: $|\Sigma^{\leq 17}| - |X| \approx 2.6 \times 10^5$. What is more, in order to avoid such incorrect hypotheses as: $(a+b)^*$, $(a+b)^*a$, $a(a+b)^*$, $a(a+b)^*a$, $(a+b)^*bb(a+b)^*$, $(a+b)^*bbb(a+b)^*$, $(a+b)^*ba$, $(ab+b+aabb)^*(ba)^*$, etc., the set of negative words has to be sufficiently large, and therefore a GI algorithm will cause to consume much more time.

We took the Synapse system as a reference method, because in all experiments [21] its authors reported results for positive and negative words being all strings in $\{a, b\}^*$ with a length not longer than $d = 7$. This is very similar to our settings. The last thing we had to choose was the way of deciding that a CFCG is correct with respect to a model. Naturally, in view of the definition of a context-free cover-grammar every output from our algorithm is correct, but some of them could be beyond what is expected. Unfortunately, for two given CFGs, G_1 and G_2, checking whether $G_1 \equiv G_2$ is undecidable (there is not even an inefficient way for doing this algorithmically). For this reason, one has to verify it formally by hand or check it by a careful examination. Since the formal proof of the grammar equivalence can be extremely difficult to find and because of multiple runs, the checking with a test set seemed to be a better choice even if it might fail.

7 Conclusions

In this paper we were interested in the synthesis of context-free grammars based on finite languages. We therefore faced the following task: given a finite set $X \subset \Sigma^+$ of words where d is the length of the longest word in X, build a context-free grammar—called a cover-grammar—that accepts the language X as well as possibly other words longer than d and does not accept any word from the set $\Sigma^{\leq d} - X$. As stated, the cover-grammar problem has many solutions among which we are searching for short ones. For regular languages, this problem turns into a minimal cover-automata problem and has many theoretical results and practical methods [3, 17]. In order to address the context-free case, we have designed an algorithm for the multi-decomposition of a finite language, in which the most important and intractable computation is aided by ant colony optimization. On this foundation, we have further designed an algorithm for generating a cover-grammar in $(\ell, 2)$-normal form, which not only is able to generate concise grammars, but also might identify infinite languages based on the finite examples X. The conducted experiments showed that our algorithm can synthesize fundamental context-free grammars within a second, which include most of the grammars investigated by other researchers.

References

[1] Bondy, J.A., Murty, U.S.R.: Graph Theory. Graduate Texts in Mathematics 244, Springer, 2008.

[2] Bunke, H., Sanfelieu, A. (eds): Syntactic and Structural Pattern Recognition Theory and Applications. World Scientific, Singapore, 1990.

[3] Câmpeanu, C., Sântean, N., Yu, S.: Minimal cover-automata for finite languages. Theor. Comput. Sci. 267, 3–16, 2001.

[4] Charikar, M., Lehman, E., Liu, D., Panigrahy, R., Prabhakaran, M., Sahai, A., Shelat, A.: The smallest grammar problem. IEEE Trans. Inf. Theory 51, 2554–2576, 2005.

[5] Chirathamjaree, C., Ackroyd, M.: A method for the inference of non-recursive context-free grammars. Int. J. Man-Machine Studies 12, 379–387, 1980.

[6] Colorni, A., Dorigo, M., Maniezzo, V.: Distributed Optimization by Ant Colonies. In Varela, F. and Bourgine, P., editors, Proceedings of ECAL91 - First European Conference on Artificial Life, Elsevier Publishing, 134–142, 1992.

[7] Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein C.: Introduction to Algorithms. MIT Press, second edition, 2001.

[8] Dréo, J., Pétrowski, A., Siarry, P., Taillard, E.: Meta-heuristics for Hard Optimization. Springer, 2006.

[9] Du, D.-Z., Ko, K.-I: Problem Solving in Automata, Languages, and Complexity. Wiley, 2001.

[10] Eyraud, R., de la Higuera, C., Janodet, J.: LARS: A learning algorithm for rewriting systems. Mach. Learn. 66, 7–31, 2007.

[11] Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman, 1979.

[12] de la Higuera, C.: Characteristic sets for polynomial grammatical inference. Mach. Learn. 27, 125–138, 1997.

[13] de la Higuera, C.: A bibliographical study of grammatical inference. Pattern Recognit. 38, 1332–1348, 2005.

[14] de la Higuera, C.: Grammatical Inference: Learning Automata and Grammars. Cambridge University Press, 2010.

[15] Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation, 2nd ed. Addison-Wesley, 2001.

[16] Kieffer, J.C., Yang, E.: Grammar Based Codes: A New Class of Universal Lossless Source Codes. IEEE Trans. Inf. Theory 46, 737–754, 2000.

[17] Körner, H.: On minimizing cover automata for finite languages in $O(n \log n)$ time. LNCS 2608, Springer, 359–400, 2003.

[18] Lothaire, M.: Algebraic Combinatorics on Words. Encyclopedia of Mathematics and Its Applications 90, Cambridge, 2002.

[19] Mateescu, A., Salomaa, A., Yu, S.: On the decomposition of finite languages. Technical Report 222, Turku Centre for Computer Science, December 1998.

[20] Moon, J.W., Moser, L.: On cliques in graphs. Isr. J. Math. 3, 23–28, 1965.

[21] Nakamura, K., Ishiwata, T.: Synthesizing context free grammars from sample strings based on inductive CYK algorithm. LNAI 1891, Springer, 186–195, 2000.

[22] Nakamura, K., Matsumoto, M.: Incremental learning of context free grammars based on bottom-up parsing and search. Pattern Recognit. 38, 1384–1392, 2005.

[23] Papadimitriou, C. H., Steiglitz K.: Combinatorial Optimization: Algorithms and Complexity. Dover Publications, July 1998.

[24] Rabhi, F., Lapalme, G.: Algorithms: A Functional Programming Approach. Addison-Wesley, 1999.

[25] Salomaa, A., Yu, S.: On the decomposition of finite languages. In Rozenberg, G., and Thomas, W., editors, Developments in Language Theory, World Scientific, 22–31, 2000.

[26] Wieczorek, W.: An algorithm for the decomposition of finite languages. Logic Journal of the IGPL 18, 355–366, 2010.

[27] Wood, D.: A generalised normal form theorem for context-free grammars. Comput. J. 13, 272–277, 1970.

Received 21.04.2016, Accepted 22.09.2016