An Online Induction Algorithm for Internal Contextual Grammars using Restarting Automata

Abhisek Midya\textsuperscript{a,}, Lakshmanan Kuppusamy\textsuperscript{b}, Sumitha V.S\textsuperscript{a}

\textsuperscript{a}Computer Science & Engineering, Christ University, Bangalore - 560 029.
\textsuperscript{b}School of Computing Science & Engineering, VIT University, Vellore - 632 014.

Abstract

In this paper, we propose a new algorithm to induce an internal contextual grammar from positive examples using restarting automata. Motivation comes from real-time systems which induce the target grammar within a deadline. In our algorithm, we deal with real time inputs which are generated by internal contextual grammar. Principally grammatical inference and grammar induction are considered equivalent but there is a slight difference, in this paper we concentrate on that difference. Here initially our algorithm will concentrate on grammatical Inference but at last it will be ended up with the concept of grammar induction. In order to induce the grammar, we first obtain insertion rules by scanning an input at a particular time unit. The insertion rules are converted into contextual rules. This set of contextual rules will be a guess about the grammar without taking care of over generalization. Further we will check the correctness of the contextual rules using restarting automata for the next input string and we update the rules based on need, that is called correction phase. After getting the final time-unit/deadline as an input, the algorithm executes some steps on the induced grammar to prune the over generalization of strings. It produces the final grammar for the strings which are given within the final time-unit.

Keywords: Restarting automaton; grammatical inference; grammar induction; internal contextual grammars; insertion grammars.

1. Introduction

The restarting automaton was introduced by Jancar et al. in 1995 in order to model the so called ‘analysis by reduction’, which is a technique used in linguistics to analyze sentences of natural languages. Restarting automaton is a type of regulated rewriting system. A restarting automaton contains a finite control unit, a head with a look-ahead window attached to a string tape. At several points it cuts-off substrings from the look-ahead window using DEL operation followed by restart (RST) operation. The head moves right along the tape until it takes any RST operation. RST means that the restarting automaton places the look-ahead window over the left border of the tape and it completes one cycle. After performing a DEL/RST operation, the restarting automaton cannot remember any

* Corresponding author. Tel.: +919972303834, +917305466952
E-mail address: abhisekmidyacse@gmail.com

\textsuperscript{c}© 2016 The Authors. Published by Elsevier Ltd. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).
Peer-review under responsibility of the organizing committee of ICETEST – 2015
step of computation that was performed earlier. Further, when each time the DEL operation is performed, the tape becomes smaller and smaller, it shows that restarting automaton are linearly space bounded.

The grammatical inference concept comes from the child language learning that how children can learn the grammar of their mother tongue from examples that the surroundings offer them. Grammatical Inference refers to the method of inferring a grammar (and possibly a target language) from data. Data can be text or informant. The difference between text and informant is that a text gives only positive examples (all strings do belong to the same language) where informant is both positive and negative examples.

A learning procedure is an algorithm which is executed on a never-ending stream of inputs. The inputs are grammatical strings, taken from a target language which is in a known class of languages. The task is to identify a grammar that generates the target language. At each point in the process, any string is given as an input to the algorithm. After each input, the algorithm produces a guess about the grammar which is eventually correct and could be unaltered when additional input strings are given.

In the model proposed by Gold [6](1967), the learning procedure involves two parties, the Learner and the Challenger (also called Teacher). The former is the party that has to identify the output grammar, while the latter has to give to the learner examples taken from the language. In addition, a critic (also called an oracle) may be used by the learner to verify the correctness of the output grammar. According to the Learner and the Challenger role and the use of a critic, learning procedure are classified in supervised, unsupervised, and semi-supervised version. Learning methods are said supervised if they use a challenger and a critic to verify the output. Learning procedure are said to be unsupervised if they only use a challenger. This means that they do not receive information from the critic about the correctness of the grammar. A third class of learning procedure, called semi-supervised learning, is halfway between supervised and unsupervised learning. In this paper we are developing a supervised learning procedure.

Gold model said that no super finite language (it contains all finite language and at least one infinite language) can be learnable in the limit from positive examples. Regular, context free, context sensitive grammars are not learnable in the limit from positive examples only.

(External) Contextual grammars are introduced by S. Marcus in 1969 [4] with a linguistic motivation in mind. Internal contextual grammars [5] produce strings starting from an axiom and in each step, the context of the form $(u, v)$ is adjoined to the string based on the certain string present as substring in the derived string. Counterpart to contextual grammar is insertion grammar which is based on ‘insertion’ operation. The Insertion operation is first considered by Haussler in [2] and based on the operation, insertion systems are introduced by L. Kari in [3]. Informally, if a string $\alpha$ is inserted between two parts $w_1$ and $w_2$ of a string $w_1w_2$ to get $w_1\alpha w_2$, we call the operation insertion.

In this paper, we develop an algorithm to induce an internal contextual grammar for a given set of input strings. The class of internal contextual languages are super finite, so it cannot be learnable in the limit from positive examples only. However, in order to learnable to the best, it is in practice to weaken some conditions. In this paper, we mitigate the condition, in which the inputs are received up to a final time-unit only is considered. Our algorithm is producing the final grammar after getting the final time unit $t_f$, but before that it can guess the grammar about the unknown language without taking care of the over generalization.

Grammar Induction is about finding a grammar that can explain the data, whereas grammatical inference relies on the fact that there is a target grammar and we want to find that out. Practically this may seem to make little difference, as in both cases what probably will happen is that a set of strings will be given to an algorithm, and a grammar will have to be produced. But whereas in grammar induction this is the actual task, in grammatical inference this is still a goal but more a way of measuring the quality of the learning method. Actually in this paper we turn our attention to real-time grammar inferring algorithm. Real time systems must produce results within certain deadlines. In our algorithm, $t_f$ is considered as the deadline. Before getting the $t_f$, we were concentrating on guessing the target grammar because we had a believe that there is a target grammar, so it is called grammatical inference. But after having the $t_f$ as an input, our task becomes actual, we need to explain the given data only, it is called grammar induction.

The proposed algorithm has the following major steps: (i) defining axiom based on the length of the input strings, (ii) defining insertion rule using axiom and the examining string, (iii) checking the correctness of the insertion rules (iv) converting correct insertion rules into contextual rules, (v) checking correctness of contextual rules for a new examining string using restarting automata, (vi) making correction and updating with new rules based on the requirement, (vii) avoiding over generalization by finding out that how many times each contextual rule is applied in each
string. Our algorithm can be used to develop a real-time grammar inferring software which deals with the real-time inputs and gives the final grammar after a certain deadline.

2. Preliminaries

Throughout the paper we are using the following notations. If \( \Sigma \) is an alphabet, then \( \Sigma^* \) is the set of all strings. For a string (also called word) \( w, |w| \) is the length of the string. \( \emptyset \) denotes empty set. If a word \( x \) is a subword of \( y \), is denoted by \( x \in \text{sub}(y) \). Also in this paper we refer contextual grammar rules as contextual rules at many places. Finally for an automaton, the language accepted by \( M \) is denoted by \( L(M) \) and for a given grammar \( \gamma \), the language generated by \( \gamma \) is denoted by \( L(\gamma) \).

Below we shall discuss the basic definition of restarting automata with delete operation (DRA), internal contextual grammars and insertion grammars.

2.1. Restarting Automata with Delete Operation Only (DRA)

A restarting automaton with delete (denoted by DR-automaton or by DRA) is \( M = (Q, \Sigma, \prec, \succ, q_0, k, \delta) \) where \( Q \) is a finite set of states, \( \Sigma \) is the input alphabet, \( \prec, \succ \) are left and right border respectively (\( \prec \neq \Sigma \) ), \( k \) is size of the look-ahead window (\( k \geq 1 \)). The transition relation \( \delta \) describes different types of transition steps which are given below. \( u' \) is assumed to be the content of the look-ahead window.

- **MVR** - \( (q', MVR) \in \delta(q, u') \), if \( M \) is in state \( q \) and sees the string \( u' \) in its look-ahead window, then this MVR step shifts the look-ahead window one position to the right and \( M \) enters into the state \( q' \) where \( \succ \notin u' \).
- **DEL** - \( (q', v') \in \delta(q, u') \), if \( M \) is in state \( q \) and sees the string \( u' \) in its look-ahead window, deleting an item from the look-ahead window. \( u' \) is replaced by its scattered substring \( v' \) such that \( |v'| < |u'| \). The border markers \( \prec, \succ \) must not disappear from the tape. After deleting, the head can still read the remaining part of the look-ahead window and also the automaton can place its head to the right of the just rewritten (deleted) string\(^1\).
- **RST** - It causes \( M \) to move its look-ahead window to the left border marker \( \prec \) and re-enters into the initial state \( q_0 \).
- **ACCEPT** - \( Accept \in \delta(q, u') \mid q \in Q \). It gets into an accepting state.
- **REJECT** - If \( \delta(q, u') = \emptyset \) (i.e., when \( \delta \) is undefined), then \( M \) will reject.

A configuration of the automaton \( M \) is \((u', q, v')\), where \( u' \in (\Sigma^* \cup \lbrace \lambda \rbrace) \) is the content of the working list from the left border till the position of the head, \( q \in Q \) is the current state and \( v' \in (\Sigma^* \prec \cup \Sigma^* \succ) \) is the content of the working list (from \( q \)) after the scanned item until the right border. In the initial configuration on an input word \( w \), the control unit is in the fixed initial state \( q_0 \in Q \), and the head is attached to the left border \( \prec \), i.e \((\lambda, q_0, w\prec\succ)\) (scanning \( \prec \) and looking at the next \( k - 1 \) symbols). We suppose that the states \( Q \) of the finite control are divided into two classes. The non-halting states (at least one instruction must be there which is applicable when the unit is in such a state) and the halting states (any computation ends by entering into such a state).

In general, restarting automaton is non-deterministic. In our paper we are using non-deterministic version of of DRA.

Any finite computation of a DRA consists of certain phases. A phase, called a cycle, starts in a restarting configuration, the head moves along the tape and performing MVR, DEL operations until a RST operation is performed and thus a new restarting configuration is reached. If no further RST operation is performed, any finite computation necessarily finishes in a halting configuration - such phase is called tail.

The notation \( u' \implies_M v' \) indicates that there exists a cycle of \( M \) starting in the initial configuration with the word \( u' \) and ending in the configuration having the word \( v' \), the relation \( \implies_M \) is the reflexive and transitive closure of \( \implies_M \).

We say that \( u' \) is reduced to \( v' \) by \( M \) if \( u' \implies_M v' \), we are certain that the word \( v' \) is strictly shorter than \( u' \) (substring was deleted during the cycle).

\(^1\) in our paper, we assume that every DEL operation is immediately followed by RST, it’s forming DEL-RST
An input word \( w \) is accepted by \( M \) if there is a computation which starts in the initial configuration with \( w \) (bounded by borders \( \langle, \rangle \) on the list and finishes in an accepting configuration where the control unit is one of the accepting states. \( L(M) \) denotes the language consisting of all words accepted by \( M \) and we say that \( M \) recognizes the language \( L(M) \).

**Normal DRA 1.** A DRA is called normal if all the DEL operations are in the form \((q', v') \in \delta(q, u')\) where \( v' \) is a scattered substring of \( u' \), there exist words \( x_1, x_2, x_3, x_4, x_5 \in \Sigma^* \) such that \( u' = x_1x_2x_3x_4x_5 \) and \( v' = x_1x_3x_5 \), that is at most two substrings of \( u' \) can be deleted in a cycle.

### 2.2. Internal Contextual Grammars

An internal contextual grammar with choice is a construct \( G = (V, A, C, \varphi) \), where \( V \) is an alphabet, \( A \) is a finite language over \( V \) called the set of **axioms**, \( C \) is a finite subset of \( V^* \times V^* \), called **contexts** and \( \varphi \) is a mapping defined as \( \varphi : V^* \rightarrow 2^C \), called selection/choice mapping. When the selectors are of a particular family of languages, the grammar \( G \) is said to be a contextual grammar with \( F \) selector (or \( F \) selection) where \( F \) is a family of languages such as **finite**, **regular** or others. In this paper we will deal with finite selectors. The usual derivation in the **internal mode** is defined as \( x \xrightarrow{\varphi} y \) iff \( x = x_1x_2x_3, y = x_1ux_2yx_3 \), for \( x_1, x_2, x_3 \in V^* \), \((u, v) \in \varphi(x_2)\). We call the argument \( x_2 \) in \( \varphi(x_2) \) as **selector** throughout this paper. The language generated by the above grammar \( G \) is given as \( L_w(G) = \{ x \in V^* \mid w \xrightarrow{\varphi} x, w \in A \} \), where \( \xrightarrow{\varphi} \) is the reflexive transitive closure of the relation \( \xrightarrow{\text{in}} \).

### 2.3. Insertion Grammars

An insertion grammar \( \gamma = (T, A, I) \), where \( T \) is an alphabet, \( A \) is a finite set of **strings** over \( T \) called **axioms**, \( I \) is a finite set of subsets of \( T^* \times T^* \) called **insertion rules**. The language \( L(\gamma) \) is generated by \( \gamma \) is defined as \( \gamma = \{ w \in T^* \mid y \in A : y \Rightarrow^* w \} \).

### 3. Inferring Grammar for Internal Contextual Language

In this section, we propose an algorithm to infer an internal contextual grammar from positive examples only where input strings are received from time to time.

This algorithm takes finite sequences of positive examples until a final time-unit \( t_f \). \( i_t \) is the set of given input strings at time unit \( t \) where \( t_1 \leq t \leq t_f \). Our goal is to find internal contextual grammar \( G \), such that \( I \subseteq L(G) \) where \( I \) is the set of input strings. The algorithm works in the following way. After receiving the input string at time \( t \), based on the string lengths, firstly the algorithm determines the axiom, then it defines insertion rules from an input string and the axiom. The algorithm then checks the correctness of the insertion rule. After that, correct insertion rules are converted into contextual rules which will be a guess (denoted by \( g_n \)) about the unknown grammar at time \( t_1 \). It updates with new the contextual rules if the next input string cannot be generated by the existing contextual rules. All the guessing will be done in a flexible way in the sense that the correction are done at every instance. The algorithm gives the final grammar after getting the final time-unit \( t_f \) and also takes care of the over generalization. Here after no more strings will be given as input.

Here our algorithm is the learner (teacher), challenger supplies the real time inputs and the critic is restarting automaton who verifies the correctness of the set of contextual rules. So we have developed a supervised learning procedure which consists of learner with critic and challenger.

Now we present the algorithm and the major steps involved in that are given as subsections.

Let \( i_t \) be the set of input strings at time-unit \( t \). \( i_t = \{ s_1, s_2, \ldots, s_r \} \) where \( s_j = s_{j1}s_{j2} \ldots s_{jr}, 1 \leq j \leq k, 1 \geq r \). (i.e., \( S_j \) is of length \( r \)).

#### 3.1. Finding Axiom

In order to find the axiom, the length of each string is evaluated. The smallest string will be considered as an axiom. \( \text{Axiom} = \{ s^d_j \mid s^d_j = \min|s_j| \} \). If two strings are given with same length then both will be in the axiom set \( A \). At
any point of time a string can be given as an input which is smaller than some members of the existing axiom set. In such cases, if the longer strings existing in the axiom set can be generated from this new smaller string, then this new smaller string will replace those longer strings. If no member of the existing axiom set can be generated from the new smaller string then the new smaller string will be added to the axiom set. Here axiom can be evaluated in polynomial time.

3.2. Defining Insertion Rule and Converting it into Contextual Rule:

We now shortly describe about the intuitive idea of the below steps 1-4. After axiom is identified, an examining string is chosen randomly. We now try to identify the selectors from the axiom and contexts from examining string. Thus, from step 1 to step 4, we are identifying the contextual rules that will derive \( s'_j \) from axiom.

- **Let the axiom be** \( s''_j = s''_1 s''_2 s''_3 \ldots s''_n \) and the examining (scanning) string be \( s'_j = s'_1 s'_2 \ldots s'_{j_r} \) where \( r \) = length of the examining string.

- **step 1:** let the initial rule be \( (u, x, v)_{\text{ins}} \mid u = s''_j, v = s''_2 s''_3 \ldots s''_n \), check whether any \( |x| \leq r \) exists with \( u x v \in \text{sub}(s'_j) \) or not. If yes then fix that \( x \) (i.e., substuting that \( x, u x v \) will be a subword of \( s'_j \)) and go to step 3. Else, go to step 2. For one insertion rule it takes liner time.

- **step 2:** Remove the last alphabet of the right context \( v \) and the rule becomes \( (u, x, v)_{\text{ins}} \mid u = s''_1, v = s''_2 s''_3 \ldots s''_{n-1} \). Check whether any \( |x| \leq r \) exists with \( u x v \in \text{sub}(s'_j) \) or not, if yes, go to step 3. Else, go to (recursively) step 2 until the rule becomes of the form \( (u, x, v) \mid u = s''_1, v = s''_2 \). Then go to step 4. Step 2 and step 3 take same time.

- **step 3: Conversion into Contextual Rule** After getting correct insertion rules (which necessarily satisfy \( u x v \in \text{sub}(s'_j) \) ), they are converted into 1-sided\(^2\) contextual rules. Here we are using restarting automata to check the correctness of the contextual rules, it indicates that using restarting automata the membership\(^3\) problem can be solved where the contextual grammars are given with 1-sided rules. An insertion rule can be converted into contextual rule in \( O(1) \).

Once the insertion rules are fixed, they are converted into contextual rules as follows: \( (u, x, v)_{\text{ins}} \rightarrow (u, (\lambda, x))_{\text{crg}} \) and the omitted right context \( v_{\text{ins}} \) will be treated as the left context \( u_{\text{ins}} \) for the next insertion rule. Now, we remove the \( u_{\text{ins}} \) and \( x \) from the examining string and only \( u_{\text{ins}} \) from the axiom. Thus, axiom and the examining strings are modified and now go to step 1 again. This is repeated until the axiom is covered (or scanned) completely and all selectors are obtained. The right context of the last insertion rule is omitted at the time of converting into contextual rule but can be used later as a selector in the correction and updating part in section 3.4. After getting all the contextual rules, we will check the correctness of rules for the next input string, using restarting automaton. (See section 3.3).

- **step 4:** \( u_{\text{ins}}, v_{\text{ins}} \) of the last version will be assigned together as a new left context, \( u_{\text{ins}} = s''_1 s''_2 \). Rest of the axiom part will be assigned to right context \( v_{\text{ins}} \), of the new rule, and the new rule becomes \( (u, x, v)_{\text{ins}} \) where \( u = s''_1 s''_2, v = s''_3 \ldots s''_n \) and go to step 1 until \( u_{\text{ins}} \) becomes \( s''_1 s''_2 s''_3 \ldots s''_n \). In that case, defining insertion rule is not possible. It also takes linear time for one insertion rule.

Ultimately this total subsection (Defining Insertion Rule and Converting it into Contextual Rule) can be done in \( O(n^k) \) where \( k \) is an integer. Before we move to next process, we present an example to highlight the major steps along with the discussion going on. Another example is provided in Appendix.

Now, we will check whether the next input string (i.e., new examining string) can be generated (or equivalently derived) by the existing contextual rules \( R_1, R_2, R_3 \) using the idea of membership problem for contextual grammars. This is same as the solving the membership

\(^2\) in an 1-sided contextual rule either left context \( u = \lambda \) or right context \( v = \lambda \)

\(^3\) The membership problem for a language is defined as given a grammar \( G \) and a string \( w \), whether \( w \) belongs to the language generated by \( G \) or not?
problem and we adopt the idea used in [1]. If the new examining string is not derivable, then we go for correction and updating with new rules (see Section 3.4).

3.3. Checking Membership Problem of the Input String using Restarting Automata:

Now, given a string, we shall see that how the restarting automaton is used to simulate the derivation of contextual grammar in reverse order. Let \( g_{tk} \) be the guess of a contextual grammar at time-unit \( t_k \) where \( k \geq 1 \), \( g_{tk} = R \) where \( R \) is a set of contextual rules.

In contextual grammars, contexts are only adjoined left and right to a selector string. Normal DRA simulates the derivation of contextual grammar in the reverse order. In a normal DRA, \( w \) is given as an input. It checks the string of the look-ahead window with the existing contextual rules that any of the rule \( R_i \) has been found as a substring or not. If the automaton finds \( R_i : ((sel_i, (u_i, v_i))_{icg} \in sub(u') \) where \( i \geq 1 \), \( u' = \) look-ahead window, then it deletes the left and right context \( u, v \) of that rule from the input string and takes the RST operation, otherwise takes MVR and checks whether any rule can be found as a substring in the look-ahead window or not. In this way, if the input string can be reduced back to axiom using restarting automaton then it shows that the string \( w \) can be generated using existing set of contextual rules, thus \( w \in L(G) \).

Size of Look-ahead Window of Restarting Automata:

For checking membership problem of the input string, the size of look-ahead window of \( M \) is \( k = \text{max}[|R_{\text{max}}|, k_h] + 2 \) where \( |R_{\text{max}}| \) is the maximum length of the contextual rule. \( k_h \) is the maximum axiom size i.e., \( k_h = \text{max}[|z| : z \in A] \).

The reason for 2 is added with \( k_h \) is to satisfy the accepting condition - Accept \( \in \delta(q, u') \) | \( u' = \langle \rangle, \langle z \rangle \in A \).

- ACCEPT - Accept \( \in \delta(q, u') \) | \( u' = \langle \rangle, \langle z \rangle \in A \).
- REJECT - \( \delta(q, u') = 0 \). That is when \( \delta \) is undefined. In other words, when normal DRA is unable to take any of the DEL, MVR operations then the transition becomes undefined.

Here as we are using non-deterministic version of restarting automaton, it takes exponential time.

3.4. Making Correction and Updating Rules

Below we have discussed that if the new examining string is not derivable with the existing set of contextual rules, then we need to go for correction and updating with new rules.

Let the rule be \( R_i : (sel_i, (u_i, v_i))_{icg} \) | \( u_i = \lambda \). Examining string \( s'_j = s'_1s'_2 \ldots s'_j \) \( s'_{j+1} \ldots s'_w sel_{i+1} \) \( Z \) where \( X, Z \in \Sigma^* \). \( X, Z \) are the remaining part of the string. The examining string is presented in this form \( X \select s'_j sel_{j+1} s'_j \ldots s'_w sel_{i+1} \) \( Z \) because we make the correction of \( R_i \) using \( R_{i+1} \), so it is needed to introduce the \( sel_i \) and \( sel_{i+1} \). \( sel_i = s'_j \ldots s'_j \) \( sel_{i+1} \). If selector \( sel_i, sel_{i+1} \) are not present in \( s'_j \) then new insertion rule has to be defined again to find out the correct selectors and go to section 3.2. This time no need to take care of the contexts. If defining insertion rule is not possible even after this step, then it indicates that the chosen axiom is wrong.

In that case, we choose some other axiom, if available. If no other axiom is available then we add the examining string into the axiom set (recall that we have positive examples only).

If \( v_i \neq s'_j \ldots s'_j \), then correction and updating is required. Let \( v_i \) be \( V_1V_2 \ldots V_w \) and \( s'_j \ldots s'_j \) be \( D_1D_2 \ldots D_z \) for convenience sake. To apply the rule \( R_i \), properly, \( v_i \) should be matched with \( s'_j \ldots s'_j \) \( D_1D_2 \ldots D_z \) and we have shown that the correction part for one rule, in the same way can make the correction for other rules.

- case A - If the analysis starts with equality:
  - \( D_1 = V_1, D_2 = V_2 \ldots D_z = V_z \), and \( D_{i+1} \neq V_{i+1} \) or \( f = z \) or \( s = w \), then we have the following four cases.
  - case 1: If \( f = z \) and \( s = w \), it implies that matching is correct, so no need to make any correction for this rule.
  - case 2: If \( f = z \) and \( s < w \), then \( R_{i'} : (sel_{i'}, (u_{i'}, v_{i'}))_{icg} \) \( v_{i'} = D_1D_2 \ldots D_{i'}, u_{i'} = A, sel_{i'} = sel_i \). Here \( s < w \), it implies that \( v_i \) is not covered completely so the another
rule will be $R_{i+1}': (sel_{i+1}', (u_{i+1}', v_{i+1}'))_{icg} | u_{i+1}' = V_{f+1}V_{f+2}...V_{n}, v_{i+1}' = \lambda, sel_{i+1}' = sel_{i+1}$.

$$v_{f} = D_{1}D_{2}...D_{f}, u_{f} = \lambda, sel_{f} = sel_{i}.$$  

- **case 2** - If the analysis starts with inequality:
  - $D_{2} = V_{n}, D_{2-1} = V_{n-1}...D_{2} = V_{f},$ and $D_{f-1} \neq V_{f-1}$ or $s = 1$ or $f = 1$, then we have following three cases.
  - **case 1:** If $s = 1, f > 1$. $R_{f} : (sel_{f}, (u_{f}, v_{f}))_{icg} | u_{f} = V_{1}V_{2}...V_{n}, v_{f} = \lambda, sel_{f} = sel_{i+1}$. $R_{i+1} : (sel_{i+1}', (u_{i+1}', v_{i+1}'))_{icg} | v_{i+1}' = D_{1}D_{2}...D_{f-1}, u_{i+1}' = \lambda, sel_{i+1}' = sel_{i}$.  
  - **case 2:** If $s > 1, R_{s} : (sel_{s}, (u_{s}, v_{s}))_{icg} | u_{s} = V_{1}V_{s+1}...V_{n}, v_{s} = \lambda, sel_{s} = sel_{i+1}$. $R_{i+1} : (sel_{i+1}', (u_{i+1}', v_{i+1}'))_{icg} | v_{i+1}' = D_{1}D_{2}...D_{f-1}, u_{i+1}' = \lambda, sel_{i+1}' = sel_{i}$.  
  - **case 3:** If here also started from the inequality then two rules will be formed, $R_{f} : (sel_{f}, (u_{f}, v_{f}))_{icg} | v_{f} = V_{1}V_{2}...V_{n}, u_{f} = \lambda, sel_{f} = sel_{i}$. $R_{i+1} : (sel_{i+1}', (u_{i+1}', v_{i+1}'))_{icg} | u_{i+1}' = V_{1}V_{2}...V_{n-1}, sel_{i+2} = sel_{i}$.

- If any rule $(sel, (u, \lambda))$ is incorrect where $sel$ is the first selector then no need to make the correction, we will keep the existing rule. At the time of checking the correctness of the contextual rules, if we found that any context is present in the left of the selector then we will form a new external contextual rule in the previous manner.

After analyze the full procedure, we can conclude that modification and updating of rules can be done in polynomial time.

### 3.5. Controlling over generalization after getting Final Time Unit ($t_{f}$)/Deadline

Now, if at any point of time we get a final time unit ($t_{f}$) then we have to take care of the over generalization of the contextual rules to produce the restricted form of grammar. We will arrange all the rules based on their context size in increasing order. Next we try to apply the rules according to the order and find out that how many times each rule is used in each string. It is presented in table. If it is found that without using any rule we can generate all strings then we can ignore that rule. Actually all the final rules are 1-sided where left contexts or right contexts are null that generates more strings. Thus, to control this over generalization, we check that how many times each rule is applied in each string.

### 4. Conclusion

In this paper, we have proposed an online learning algorithm for internal contextual grammar using restarting automaton. In our algorithm all major steps take polynomial time except checking the correctness of contextual rules as we are using the non-deterministic restarting automaton. We have only initiated the work in inferring contextual grammars and look for the possibility to run it in polynomial time.

There is also scope for future work. The merging concept we discussed is only the beginning step to get a better contextual grammar (in the form of two-sided contexts). A few more merging techniques can be adopted which will simplify the output of the grammar.

### References

[1] P. Jancar, F. Mraz, M. Plátek, M. Procházka, J. Vogel: Deleting Automata with a Restart Operation; In: proceedings of Developments in Language language theory III(97), Ed. S.Bozapalidis, Thessaloniki, pp. 191202, 1998.

[2] D. Haussler: Insertion and Iterated Insertion as Operations on Formal Languages, Ph.D. Thesis, University of Colorado, Boulder, 1982.

[3] L. Kari, G. Thierrin: Contextual Insertions/Deletions and Computability, Information and Computation 131 1, pp. 47-61, 1996.
[4] S. Marcus: Contextual Grammars, *Revue Roumaine de Mathematiques Pures et Appliques* **14**, pp. 1525-1534, 1969.

[5] Paun, Gh., Nguyen, X.M.: On the Inner Contextual Grammars, *Rev. Roum. Pures. Appl.*, **25**, pp. 641-651, 1980.

[6] E.M. Gold, Language Identification in the Limit, *Inf. and Control*, **10**, pp. 447-474, 1967.