JVLC

Journal of Visual Language and Computing

Volume 2019, Number 1

Journal of Visual Language and Computing

journal homepage: www.ksiresearch.org/jvlc/

Context Computation for Implicit Context-Sensitive Graph Grammars: Algorithms and Complexities

Yang Zou[a], Xiaoqin Zeng, and Yufeng Liu

Institute of Intelligence Science and Technology, School of Computer and Information, Hohai University, China

ARTICLE INFO

Article History:
Submitted 3.1.2019
Revised 6.1.2019
Second Revision 7.22.2019
Accepted 8.26.2019

Keywords:
Visual languages
Context-sensitive graph grammars
Context computation
Algorithm
Complexity

ABSTRACT

Visual Programming Languages have been frequently utilized in computer science. Context-sensitive graph grammars are appropriate formalisms for specifying visual programming languages, since they are intuitive, rigorous, and expressive. Nevertheless, some of the formalisms whose contexts are implicitly or even incompletely represented in productions, called implicit context-sensitive graph grammars, suffer inherent weakness in intuitiveness or limitations in parsing efficiency. Making context explicit to productions tends to be a conceivable way to address this issue. Based on the formalization of context, this paper proposes an approach to the computation of context for implicit context-sensitive graph grammars. The approach is comprised of four partially ordered algorithms. Moreover, the complexities of the algorithms are analyzed and the applicability of the approach is discussed. Thus, the proposed approach paves the way for the practical applications of context in implicit context-sensitive graph grammar formalisms, such as facilitating the comprehension of graph grammars and improving parsing performance of general parsing algorithms.

1. Introduction

In many fields of computer science, Visual Programming Languages (VPLs) have been frequently adopted in modeling, representation, and design of complex structures. VPLs usually handle those objects that do not possess inherent visual representation in a visual way [1].

Various approaches have been proposed to formally specify and parsing VPLs, such as constraint multiset grammars [2], symbol-relation grammars [3], picture processing grammar [4], visual grammar [5], attributed shape grammar [6], compiler techniques [7], etc. As a natural extension of formal grammar theory, graph grammars offer the mechanisms for formal specification and parsing of VPLs [8], just like formal grammars do for string languages. However, the extension from one-dimensional string-based formal grammars to two-dimensional graph grammars brings about a few novel challenges, especially the embedding problem. The embedding problem refers to that how to avoid creating dangling edges when replacing a subgraph in a graph (called host graph) with another graph and connecting the remainder and the replacing graph together to produce a whole graph. Quite a few graph grammar formalisms have been proposed in the literature [8-12]. From the perspective of usability, there is still room for these formalisms to be ameliorated in expressive power or computing efficiency.

Most of the existing graph grammar formalisms fall into the categories of context-free and context-sensitive. The expressive power of a graph grammar lies on the type it belongs to as well as the embedding mechanism it chooses [13-14]. Among all the categories of embedding mechanisms that vary in complexity and power, invariant embedding is the least complex one and most commonly

[a]Corresponding author
Email address: yzou@hhu.edu.cn

DOI reference number: 10.18293/JVLC2019N1-017

employed in graph grammar formalisms. Context-sensitive graph grammars tend to be more expressive than context-free ones, when confined to identical less complex embedding mechanisms and invariant embedding in particular. As context-free graph grammars have difficulty in specifying a large portion of graphical VPLs [11-12], recent research in this field focus more on context-sensitive graph grammar formalisms and their applications [15-22].

Generally, a graph grammar consists of a set of productions (rewriting rules), each of which is a pair of graphs, called left graph and right graph, together with an embedding expression. In context-sensitive graph grammars, the contexts pertaining to a production generally refer to the neighboring subgraphs of the rewritten portion of its left graph in potential host graphs [23], which describe the situations under which the production can be applied. A host graph is a graph that is being rewritten by some graph grammar in the process of derivation or parsing. However, the context portion of a production, i.e., the remainder of the left graph minus the rewritten portion, is commonly not a direct copy of the contexts for the sake of conciseness of productions and easiness of embedding.

As is known, the most representative context-sensitive graph grammar formalisms are Layered Graph Grammar (LGG) [11] and Reserved Graph Grammar (RGG) [12]. In order to solve the embedding problem, LGG identically involves in the left and right graphs of a production its immediate context and imposing a dangling edge condition on redex definition, which guarantees that dangling edges never occur in rewritten host graphs. Generally, a redex is a subgraph in a host graph that is isomorphic to the left or right graph of a production. RGG is commonly viewed as an improvement over LGG in respect of succinctness of specification and efficiency of parsing algorithm. Rather than directly involving contexts in productions just as LGG does, RGG formalism invents a particular two-level node structure coupled with a marking technique to indirectly specify the context of a production by identically distributing a set of marked vertices into the left and right graphs. The vertices establish a one-to-one correspondence between the two graphs in terms of their marks. Thus, the embedding problem is solved through this mechanism together with a dedicated embedding rule. Other context-sensitive formalisms include Edge-based Graph Grammar (EGG) [23-24], Context-Attributed Graph grammar (CAGG) [25], Contextual Layered Graph Grammar (CLGG) [26], and Spatial Graph Grammars (SGG) [27]. To tackle the embedding problem, EGG identically augment a set of marked dangling edges to both the left and right graphs of a production, whereas CAGG introduces attributes of nodes to establish a correspondence between the two graphs of a production. CLGG and SGG are extensions of LGG and RGG, respectively. Based on LGG, CLGG supports three extra mechanisms, which can be employed to define more complex VPLs. SGG extends RGG by augmenting its productions with a spatial specification mechanism, with which it can explicitly describe both structural and spatial relationships for VPLs.

According to how the context portion of a production is dealt with, the preceding formalisms fall into two categories: explicit and implicit [28]. The former formalisms indicate those that directly enclose the complete immediate contexts as its context portion in a production, whereas the latter refers to the ones in which the context portion is expressed as specifically tailored (i.e., incomplete) immediate contexts, attributes adhered to the rewritten portion, or even newly introduced graph notations. Apparently, LGG and RGG are typical examples of the former and the latter, respectively.

One of the inherent deficiencies of implicit formalisms is that they are not intuitive, which arises from the fact that the context portion of a production is not the complete immediate contexts. In RGG, the context portion is a set of marked vertices. Vertices are explained to be connecting points of edges, but their exact meaning is left undefined. Therefore, the selection, arrangement and marking of vertices within a node become a challenge in the design of productions. Moreover, as actual immediate contexts are absent in productions, it is rather difficult for users to exactly comprehend the language of a given graph grammar. Noticeably, similar situations also arise in other implicit formalisms.

Making context explicit can be a conceivable way to address the above issues. Obviously, explicit context is helpful in several application scenarios. Context can be employed to make up the deficiency in intuitiveness so as to facilitate the comprehension and design of implicit graph grammars. Furthermore, context can be utilized to reduce the search space in general parsing algorithms by decreasing the times of backtracking through context matching, thus improving the parsing efficiency. In the literature [28], a formal definition of context is presented and the properties are characterized, which provide a solid theoretical foundation for the computation of context. Nevertheless, the formalization of context and its properties is complex, a direct approach for computation is not available. Therefore, an explicit and detailed method for context computation is apparently a necessity for serving the purpose of context usage in application scenarios.

In this paper, on the basis of RGG formalism, an approach to context computation in implicit graph grammar formalisms is proposed. This is a subsequent research work on context, and the technical contributions

DOI reference number: 10.18293/JVLC2019N1-017

are as follows: It presents a concrete approach for context computation, which is comprised of four partially ordered algorithms with each one being dependent on its predecessors. Moreover, it provides the time complexities of the algorithms. Besides, it discusses the applicability of the approach. This method can be generalized to be applicable to other implicit formalisms. Hence, it paves the way for the application of context for the implicit context-sensitive graph grammar formalisms.

The remainder of the paper is organized as follows: Section 2 reviews the RGG formalism and excerpts the formal definition of context. Section 3 proposes an approach that consists of four algorithms to the computation of context. Section 4 addresses the complexities of the algorithms. Section 5 discusses the applicability of the algorithms. Finally, section 6 concludes the paper and proposes future research.

2. Preliminaries

A graph grammar consists of an initial graph and a collection of productions (graph rewriting rules). Each production has two graphs called left graph and right graph respectively, and can be applied to another graph called host graph. Every node in a production is either a terminal or a non-terminal node. A graph grammar defines a graph language composed of those graphs that can be derived from the initial graph by repeated applications of the productions and whose nodes are all terminal ones. A redex is a subgraph in a host graph that is isomorphic to the left or right graph of a production.

2.1 The RGG Formalism

RGG is a context-sensitive graph grammar formalism [9]. It introduces a node-edge format to represent graphs in which each node is organized as a two-level structure, where the large surrounding rectangle is the first level, called super vertex, and other embedded small rectangles are the second level, called vertices. Either a vertex or a super vertex can be the connecting point of an edge. In addition to the two-level node structure, the RGG also introduces a marking technique that divides vertices into two categories: marked and unmarked ones. Each marked vertex of a production is identified by an integer that is unique in the left or right graph where the vertex lies. A production is properly marked if each marked vertex in the left graph has a counterpart marked by the same integer in the right graph, and vice versa.

In the process of a production application, when a redex is matched in a host graph, each vertex that corresponds to a marked vertex in the left or right graph preserves its associated edges connected to nodes outside of the redex, which avoids the appearance of dangling edges during the subsequent subgraph replacement process provided that an additional embedding rule is also enforced. The embedding rule states that if a vetex in the right (or left) graph of a production is unmarked and has an isomorphic vertex in the redex of a host graph, then all the edges that are connected to the vertex should be completely inside the redex.

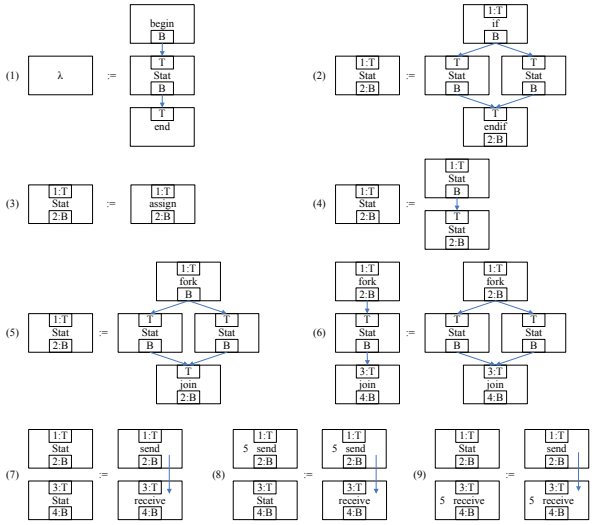

Fig. 1. A graph grammar for process flow diagrams.

As an example, an RGG specifying process flow diagrams, which is slightly adapted from [11], is depicted in Figure 1.

2.2 Partial and Total Precedence

In this subsection, we take the RGG as the representative of implicit context-sensitive graph grammar formalisms to present partial and total precedence relations between graph productions. For the sake of clarity and simplicity, some basic concepts and notations are listed below. Note that graphs are directed ones in the node-edge format and only vertices in productions might be marked.

RGG: A reserved graph grammar is a triple (A, P, Ω), where A is an initial graph, P a set of graph grammar productions, Ω a finite label set consisting of two disjoint sets Ω^T and Ω^{NT}(called terminal label set and nonterminal label set, respectively). For any production $p \coloneqq (L, R) \in P$, three conditions are satisfied: R is non-empty, L and R are over Ω, and the size of R are not less than that of L.

$p \coloneqq (L, R)$: A production with a pair of marked graphs: the left graph L and right graph R. The notations $p.L$ and $p.R$ represent the left and right graph of a production p, respectively. For any graph G, $G.N$ and $G.E$ denote the set of nodes and edges, respectively; $n.V$ and $n.v$ denote the set of vertices and some vertex v of a node n, respectively; and $G.V = \bigcup_{n \in G.N} n.V$ is the union of the sets of vertices of nodes in G; for any edge e, $s(e)$ and

DOI reference number: 10.18293/JVLC2019N1-017

$t(e)$ represent the source and target vertex of e, respectively, and $l(e)$ is the label on e. $P_L = \{p.L | p \in P\}$ and $P_R = \{p.R | p \in P\}$.

$G_1 \approx G_2$: G_1 is isomorphic to G_2.

Redex: A subgraph $X \subseteq H$ is a redex of graph G, denoted as $X \in Rd(H, G)$, if $X \approx G$ under an isomorphic mapping f and any vertex in X that is isomorphic to an unmarked vertex in G keeps its edges completely inside X.

$Rd(H, G)$: A set of redexes of marked graph G, which are subgraphs of graph H.

Mcc: A mapping from graphs to the sets of maximal connected components contained in these graphs. A maximal connected component in a graph is a connected component being maximal.

$Mcc(P_L) = \{C | C \in Mcc(p.L) \wedge p \in P\}$, $Mcc(P_R) = \{C | C \in Mcc(p.R) \wedge p \in P\}$.

The following definitions excerpted from [28] are necessary to understand the notion of context and the approach to context computation.

Definition 1. *Let $gg \coloneqq (A, P, \Omega)$ be an RGG, $p_1, p_2 \in P$ be two productions, $C_1 \in Mcc(p_1.L)$ and $C_2 \in Mcc(p_2.R)$. If $\exists X \subseteq C_2$ such that $X \in Rd(C_2, C_1)$, then C_1 is matched with X in C_2, denoted as $C_1 \approx X \sqsubseteq C_2$; or concisely C_1 is included in C_2, denoted by $C_1 \sqsubseteq C_2$.*

The definition introduces the notion of inclusion between the components of productions, or to be exact, to locate a redex of a component of the left graph of one production in some component of the right graph of another production.

Let U be some set and $S = (B, m)$ a multiset, where B is the underlying set of elements and $m: B \to \mathbb{N}$ is a mapping from B to the set $\mathbb{N}$ of positive natural numbers. $S \subseteq^{\mathbb{N}} U$ if and only if $B \subseteq U$. In order to unambiguously reference to an element from a multiset, we stipulate that any two elements in a multiset S have distinct identities even if they are the same element from the point of view of the underlying set B, and the identities of elements are not explicitly represented in context for the sake of conciseness.

Definition 2. *Let $gg \coloneqq (A, P, \Omega)$ be an RGG, and P_L and P_R the sets of left and right graphs of productions in P, respectively. A set $S_1 \subseteq Mcc(P_L)$ is included in another multiset $S_2 \subseteq^{\mathbb{N}} Mcc(P_R)$, denoted as $S_1 \sqsubseteq S_2$, if there is a mapping $f: S_1 \to S_2$ such that:*

- $\forall C \in S_1(\exists X \subseteq f(C)(X \in Rd(f(C), C)))$, *and*
- $\forall C, C' \in S_1(C \neq C' \wedge f(C) = f(C') \to \exists X, X' \subseteq f(C)(X \in Rd(f(C), C) \wedge X' \in Rd(f(C), C') \wedge X \cap X' = \phi))$.

In the definition, the first condition states that for each component in S_1, there is an image in S_2 under the mapping f that contains a redex of it; and the second expresses that if two different components in S_1 have the same image in S_2, then the two corresponding redexes in it cannot overlap, which strictly adheres to the redex definition in the RGG formalism.

Definition 3. *Let $gg \coloneqq (A, P, \Omega)$ be an RGG, and $p_1, p_2 \in P$ be two productions, p_1 directly partially precedes p_2, denoted as $p_1 \preccurlyeq_d p_2$, if $\exists S \subseteq Mcc(p_2.L)$ such that $S \sqsubseteq Mcc(p_1.R)$. The direct partial precedence relation between them is denoted by the pair $\langle p_1, p_2 \rangle$. The direct partial precedence relation on the set P of productions is defined as $\preccurlyeq_P = \{\langle p_1, p_2 \rangle | p_1, p_2 \in P \wedge p_1 \preccurlyeq_d p_2\}$. The partial precedence relation between them is denoted by the pair $\langle p_1, p_2 \rangle$.*

The direct partial precedence between a pair of productions characterizes the fact that a component of one production's left graph is isomorphic to a subgraph included in a component of another production's right graph.

The partial precedence relation is the closure of the direct partial precedence relation on a set P of productions.

Partial precedence is a kind of relation between a pair of components chosen from two distinct productions, whereas total precedence describes the same relation between two sets of components from the right graphs of a subset of productions and the left graph of a single production, respectively.

Definition 4. *Let $gg \coloneqq (A, P, \Omega)$ be an RGG, $p \in P$, and a multiset $P' \subseteq^{\mathbb{N}} P$. P' directly totally precedes p, denoted as $P' \prec_d p$, if there is a surjective mapping $f: Mcc(p.L) \to St$ such that:*

- $St \subseteq Mcc(P'_R)$;
- $Mcc(p.L) \sqsubseteq Mcc(P'_R)$ *with respect to f;*
- $\forall p' \in P'(\exists C \in Mcc(p.L)(f(C) \in Mcc(p'.R)))$.

The corresponding direct total precedence relation is denoted by the pair $\langle P', p \rangle$, and p and P' are called the target production and preceding set, respectively. The direct total precedence relations on the set P of productions is defined as $\prec_P = \{\langle P', p \rangle | P' \subseteq^{\mathbb{N}} P \wedge p \in P \wedge P' \prec_d p\}$.

A direct total precedence relation specifies that a certain graph composed of the right graphs of a subset of productions contains a redex of the left graph of another production. Note that the third constraint on f emphasizes that every production in P' takes part in f with at least one of its components in the right graph. If a subset P' of P forms such a relation with a production p, then it means

DOI reference number: 10.18293/JVLC2019N1-017

that all the right graphs of P' must exactly comprise a redex of the left graph of p with each one containing at least one redex of its components.

A total precedence relation $\langle M, p\rangle$ is composed of a set of direct total precedence relations and a set of linking relations on it. A compound precedence set consists of three parts: a multiset T of productions from set P, a multiset E of direct total precedence relations, and a set R of linking relations on E. A compound precedence set M, together with a production p, forms a total precedence relation, on condition that a direct total precedence relation is established between the first part of M and the production p.

2.3 Definition of Context

The sets of partial or total precedence relations with respect to a graph grammar establish an order of production applications, which can be exploited to discover potential situations in which any of the productions is applicable for derivation. We refer to these situations as contexts. Given two productions p_1 and p_2, if p_1 directly partially precedes p_2, then $p_1.R$ contains a context of p_2 or merely a portion of a context, depending on whether $p_2.L$ consists of only one or at least two maximal connected components. As for the former case, $\{p_1\} \prec_d p_2$ readily holds and a context of p_2 immediately follows; whereas in the latter case, a subset of productions involving p_1 that directly totally precedes p_2 is pursued so as to form a complete context for p_2. As a third case, a total precedence relation can be sought to build a rather deeper complete context.

Complete contexts of a production can be stratified in terms of the levels of corresponding total precedence relations from which they are generated. Roughly, a complete context that is built in the light of a precedence relation that corresponds to a rooted tree of depth i is called a level-i context, $i \geq 1$, and it degrades to a level-1 context when the relation is a direct one.

A complete context can be employed to extend the respective production to which it pertains. This is done by augmenting the context simultaneously to the both graphs and properly linking the two parts together respectively. A production p equipped with a level-i context is often abbreviated to a context- i p.

Definition 5. *Let* $gg \coloneqq (A, P, \Omega)$ *be an RGG,* $p \in P$, $Mcc(p.L) = \{C_1, \cdots, C_n\}$, *and* $P' \subseteq^{\mathbb{N}} P$. *If* $P' \prec_d p$ *with respect to some surjective mapping* $f: Mcc(p.L) \to St = \{D_1, \cdots, D_m\}$ *and a set of redexes* $X = \{X_i | X_i \in Rd(f(C_i), C_i), 1 \leq i \leq n\}$, *then the pair* (U, Z) *is a level-1 context of* p *with respect to* P', f, *and* X, *denoted as* $ct_p(P', f, X)$, *where* $U = \bigcup_{1 \leq j \leq m} D'_j$, $D'_j = D_j \setminus \bigcup_{k_j \in K_j} X_{k_j}$, $K_j = \{l | f(C_l) = D_j \wedge 1 \leq l \leq n\}$, $Z = \bigcup_{1 \leq j \leq m} Z_j$, *and* $Z_j = \{e \in D_j.E | (s(e) \in X_{k_j}.V \wedge t(e) \in D'_j.V) \vee (s(e) \in D'_j.V \wedge t(e) \in X_{k_j}.V) \wedge k_j \in K_j\}$. *The sets* U *and* Z *are called the contextual graph and contextual connection, respectively.*

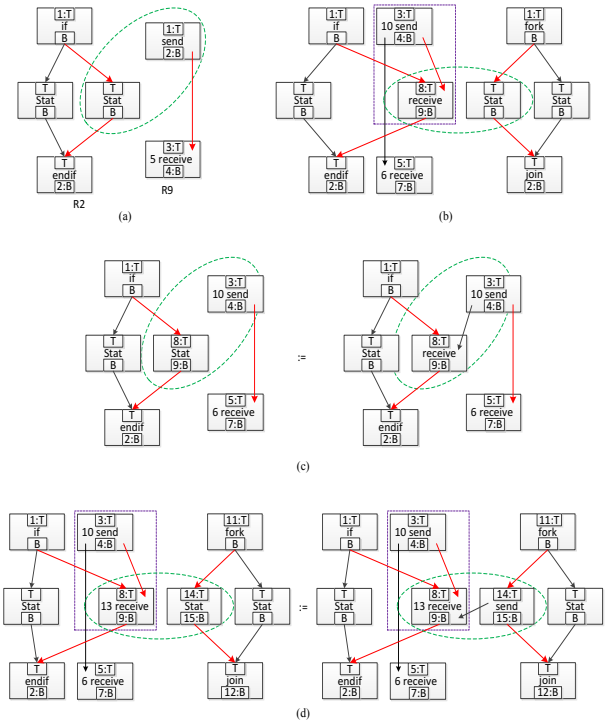

Fig. 2. The contexts of a production and their extended productions. (a) A level-1 context of $p8$. (b) A level-2 context of $p9$. (c) A level-1 context-equipped $p8$. (d) A level-2 context-equipped $p9$.

Each component D'_j of the contextual graph is the rest graph of D_j, a component of the right graph of some production that contains one or more redexes X_{k_j} of the components C_{k_j}, minus X_{k_j}, and each Z_j is the collection of edges in D_j that connect D'_j to all the redexes X_{k_j}.

Similar to Definition 5, the notion of level- i context can be recursively defined.

Example 1. Two contexts of a production and their extended productions.

Two contexts at distinct levels of a production and their respective extended productions are demonstrated in Figure 2. A level-1 context of $p8$ originated from the direct total precedence relation $\{p2, p9\} \prec_d p8$ is shown in Figure 2(a), where the left component of the graph is $R2$ (the right graph of production 2), the right one is $R9$, the subgraph enclosed by the green dashed ellipse is the redex of $L8$ (the left graph of $p8$), and the context consists of two parts U and Z: U is the rest of the whole graph

DOI reference number: 10.18293/JVLC2019N1-017

minus the redex and Z the set of thick red edges that connect U to the redex. The corresponding context-equipped production, context-1 $p8$, numbered as $p11$, is depicted in (c), where the two subgraphs surrounded respectively by green dashed eclipses are the isomorphic image of the underlying production of $p11$.

Figure 2(b) is a level-2 context of $p9$, which is created based on the direct total precedence relation $Ud(\{p5, p11\}) \prec_d p9$, or equivalently, the total precedence relation $Ul(\{p5, p11\}) \prec p9$, where $Ud(P)$ indicates the set of underlying productions of P, and $Ul(P)$ refers to the underlying structure of P. In this graph, the left and right component is $R11$ and $R5$, respectively, the subgraph enclosed by the purple dashed rectangle is the isomorphic image of $Ud(p11)$, and the one surrounded by the green dashed eclipse is the redex of the left graph of $p9$. In a similar way, the corresponding context-equipped production, context-2 $p9$, is illustrated in (d).

3. Context Computation

In this section, an approach is presented for the computation of context in the RGG formalism, based on the theoretical foundation reviewed in the preceding section.

The approach consists of four partially ordered algorithms, of which the first three deal with the computation of the set of direct partial precedence relations, the set of direct total precedence relations, and the set of total precedence relations with respect to a set of productions, respectively, and the last handles the computation of the contexts of a single production as well as the corresponding extended productions.

3.1 Computation of Partial Precedence

Algorithm 1. Computation of partial precedence relations.

Input. A set P of productions.
Output. The direct partial precedence relation on P.
{
$Cm_L = \bigcup_{p \in P} Mcc(p.L)$;
$Cm_R = \bigcup_{p \in P} Mcc(p.R)$;
Create a two-dimensional array M whose two indices range over Cm_L and Cm_R respectively such that each element is initialized to an empty set $\emptyset$;
for each $C \in Cm_L$ {
for each $C' \in Cm_R$ {
$M[C, C'] = \text{FindRedex}(C', C)$;
}
}
return M;
}

The first algorithm generates the partial direct precedence relation on a given set of productions. It consists of collecting all the components of the left and right graphs of the productions to create one set and another respectively, and then finding all the redexes of each component of the former in any one of the latter that is taken as the host graph. The output is a matrix such that each entry is assigned to a set of redexes (maybe empty) with respect to a pair of components from the two distinct sets that uniquely locate the entry in it. The function $\text{FindRedex}(C', C)$ returns all the redex of C found in C'.

For example, consider the RGG in Figure 1, the set of direct partial precedence relations regarding $p8$ is the set that is comprised of the following elements: $\langle p9, p8\rangle$, $\langle p8, p8\rangle$, $\langle p7, p8\rangle$, $\langle p2, p8\rangle$, $\langle p4, p9\rangle$, $\langle p5, p9\rangle$, $\langle p6, p9\rangle$, $\langle p1, p9\rangle$, $\langle p8, p9\rangle$. This set also coincides with the set of partial precedence relations of $p8$.

3.2 Computation of Direct Total Precedence

The second algorithm figures out the direct total precedence relation on a set P of productions, on the basis of the output of the first algorithm.

First, it creates three one-dimensional arrays Llt, Rlt, Tpd and Fun which share a same index that ranges over P, to store the upcoming data.

Then, it arranges the components of the left graph of each production p into an ordered tuple form $Llt[p]$ in a certain order, and for each element in this tuple, it generates a corresponding set that gathers all the redexes involved in the components from Cm_R.

Next, it conducts the Cartesian product $Rlt[p]$ of these sets of redexes in the same order as their counterparts in $Llt[p]$. As a result, every tuple in $Rlt[p]$ is a redex of $Llt[p]$, since they are of the same length and each constituent of the former is a redex of the element of the latter to which it corresponds in terms of the tuple order.

After that, by replacing each redex in a tuple of $Rlt[p]$ with the underlying production whose right graph includes a component that contains this redex, it acquires a direct total precedence relation regarding p; this process repeats until all the relations regarding p, which comprise the set $Dtp[p]$, are collected. Meanwhile, it establishes a mapping h from $Dtp[p]$ to a partition of $Rlt[p]$ (i.e., a collection of disjoint nonempty subset of it that have it as their union) such that h maps each tuple to the subset of $Rlt[p]$, each of whose elements can be transformed to this tuple by the preceding replacement.

Note that each element in $Dtp[p]$ is a list of a multiset of productions in a predefined order. Nevertheless, a direct total precedence relation refers to a relation between a multiset of productions without any order and a production. To bridge this gap, it performs a partition of $Dtp[p]$ in terms of such an equivalence relation that two lists are equivalent if both of them correspond to a same

DOI reference number: 10.18293/JVLC2019N1-017

multiset, which produces $Dps[p]$ and l.

Algorithm 2. Computation of direct total precedence relations.

Input. A set P of productions and the direct partial precedence relation on P.
Output. The direct total precedence relation on P.
{
Let Llt, Rlt, Dtp, Dps, $Fun1$ and $Fun2$ be one-dimensional arrays that share the same index which ranges over P;
for each $p \in P$**{**
$k = |Mcc(p.L)|$;
Let $Mcc(p.L) = \{C_1, \cdots, C_k\}$;
$Llt[p] = (C_1, \cdots, C_k)$;
for each $C \in Mcc(p.L)$**{**
$Rdx(C) = \emptyset$;
for each $C' \in Cm_R$**{**
$Rdx(C) = Rdx(C) \cup M[C, C']$;
}
}
$Rlt[p] = Rdx(C_1) \times \cdots \times Rdx(C_k)$;
$Dtp[p] = \emptyset$;
for each k-tuple $(t_1, \cdots, t_k) \in Rlt[p]$**{**
Generate a k-tuple $(p_1, \cdots, p_k)$ such that
$t_i \in \text{FindRedex}(Mcc(p_i.R), C_i)$, $1 \le i \le k$;
if $((p_1, \cdots, p_k) \notin Dtp[p])$**{**
$h((p_1, \cdots, p_k)) = \{(t_1, \cdots, t_k)\}$;
$Dtp[p] = Dtp[p] \cup \{(p_1, \cdots, p_k)\}$;
} ***else***
$h((p_1, \cdots, p_k)) = h((p_1, \cdots, p_k)) \cup \{(t_1, \cdots, t_k)\}$;
}
$Fun1[p] = h$;
$Dps[p] = \emptyset$;
for each list $(q_1, \cdots, q_k) \in Dtp[p]$**{**
Create a multiset $S = \{q_1, \cdots, q_k\}$;
if $(S \notin Dps[p])$**{**
$l(S) = \{(q_1, \cdots, q_k)\}$;
$Dps[p] = Dps[p] \cup \{S\}$;
} ***else***
$l(S) = l(S) \cup \{(q_1, \cdots, q_k)\}$;
}
$Fun2[p] = l$;
}
return $(Llt, Rlt, Dtp, Dps, Fun1, Fun2)$;
}

In this way, it finally achieves the arrays Dps and $Fun2$ that can generate the direct total precedence relations regarding each production of P, together with the array $Fun1$ that can produce the set of redexes with respect to any of these relations. For example, for each $S \in Dps[p]$, $\langle S, p \rangle$ is a direct total precedence relation, and $\bigcup_{Q \in l(S)} h(Q)$ is the set of all the possible redexes.

An underlying assumption for the algorithm is that any component in Cm_L or Cm_R, as well as any redex of a component from Cm_L in another from Cm_R, is uniquely identified. This is applicable from the perspective of algorithm implementation, as it can be achieved by assigning to each node of a production a unique number as its identity, and representing each redex as a triple (S, f, id) with S the involved subgraph, i.e., the redex itself, f the underlying mapping, and id the identifier of the right graph that contains the redex.

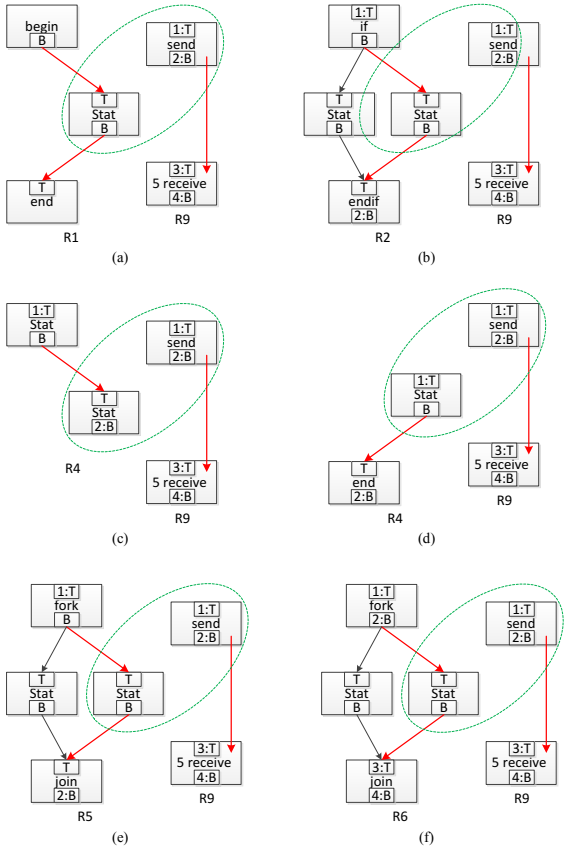

Fig. 3. Some level-1contexts of $p8$.

For example, consider the RGG illustrated in Figure 1, the direct total precedence relations with $p8$ being the target production are as follows: $\langle \{p1, p9\}, p8 \rangle$, $\langle \{p2, p9\}, p8 \rangle$, $\langle \{p4, p9\}, p8 \rangle$, $\langle \{p5, p9\}, p8 \rangle$, $\langle \{p6, p9\}, p8 \rangle$, $\langle \{p1, p8\}, p8 \rangle$, $\langle \{p2, p8\}, p8 \rangle$, $\langle \{p4, p8\}, p8 \rangle$, $\langle \{p5, p8\}, p8 \rangle$, $\langle \{p6, p8\}, p8 \rangle$, $\langle \{p1, p7\}, p8 \rangle$, $\langle \{p2, p7\}, p8 \rangle$, $\langle \{p4, p7\}, p8 \rangle$, $\langle \{p5, p7\}, p8 \rangle$, $\langle \{p6, p7\}, p8 \rangle$. The preceding set of the relations includes two productions, each of whose right graph contains a component, and these two components constitutes a graph that contains a redex of the target production's left graph. Therefore, the number of total precedence relations regarding $p8$ is $5 \times 3 = 15$. The algorithm figures out all the total precedence relations with any production from the input production set assuming the role of target production.

Figure 3 illustrates some of the level-1 contexts regarding $p8$. These six level-1 contexts correspond to the first five direct total precedence relations with $p8$ being the target production. Note that $\langle \{p4, p9\}, p8 \rangle$ accounts for two of them, i.e., Figure 3(c) and (d), as there are two options for the selection of node "Stat". Readily, the total number of level-1 contexts corresponding to the direct

DOI reference number: 10.18293/JVLC2019N1-017

total precedence relations regarding $p8$ can be counted as $6\times3 = 18$.

3.3 Computation of Total Precedence

The third algorithm describes the procedure of constructing all the total precedence relations of depth $k+1$ based on those of depth no more than k, with respect to a set P of productions, where $k \geq 1$. Readily, the set of total precedence relations of any depth can be recursively generated from the fundamental set of direct total precedence relations by using it.

Algorithm 3. Computation of total precedence relations.

Input. A set P of productions, $\preccurlyeq_P$ the direct total precedence relation on P, and the set Tpd of total precedence relations of depth no more than k, where $k \geq 1$.
Output. The set of total precedence relations of depth $k+1$.

```
{
  Let Ptp and Tpr be one-dimensional arrays whose indexes range
    over P;
  Let Tpd_k be the set of total precedence relations of depth k;
  for each p ∈ P{
    Ptp[p] = {null}; // Initialize the elements of Ptp;
  }
  for each ps ∈ Tpd{
    Let e = ⟨P', p'⟩ be the root element of ps;
    Ptp[p'] = Ptp[p'] ∪ {ps};
  }
  for each p ∈ P{
    Ptr[p] = ∅;
    for each e' = ⟨P'', p⟩ ∈≼_P{
      Let P'' = {p_1, ···, p_k}; //P'' is a multiset;
      Crt = Ptp(p_1)×···×Ptp(p_k);
      E = {e'};
      R = ∅;
      for each (t_1, ···, t_k) ∈ Crt such that
      ∃i,j(t_i ≠ null ∧ t_j ∈ Tpd_k){ //1 ≤ i ≤ k;
        Create a mapping f with domain P'';
        D = ∅; // D is a multiset;
        for each t_i{ //1 ≤ i ≤ k;
          if (t_i = null)
            f(p_i) = null;
          else {
            Let t_i = (E_i, R_i), and e_i be the root element;
            f(p_i) = e_i;
            D = D ∪ {e_i};
            E = E ∪ E_i;
            R = R ∪ R_i;
          }
        }
        Construct a linking relation r = (e', D, f);
        R = R ∪ {r};
        Ptr[p''] = Ptr[p''] ∪ {(E, R)};
      }
    }
  }
  return Tpr;
}
```

The algorithm consists of two tasks. First, it partitions the set Tpd into $|P|$ distinct subsets in terms of the root node of the rooted tree that each total precedence relation (i.e., a precedence structure) forms, which comprise the set Ptp.

Then, for any production p in P, it constructs the set of all the total precedence relations of depth $k+1$ whose root elements share the same head p. More precisely, this set is the union of the subsets, each of which includes those relations whose root elements are a same direct total precedence relation with p as the head. Thus, in terms of each of these relations, say $\langle P'', p\rangle$, the algorithm constructs a corresponding subset of all the relations of depth $k+1$ with $\langle P'', p\rangle$ as the root element.

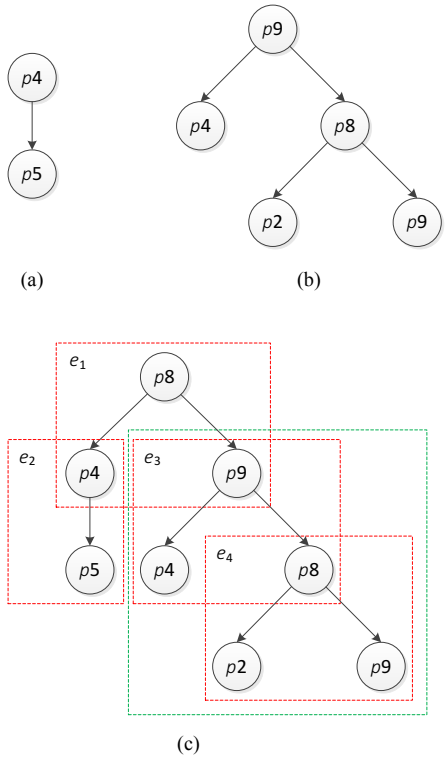

Fig. 4. The rooted trees of different depths that correspond to precedence structures.

To this end, it first conducts the Cartesian product of $|P''|$ selected elements from Ptp whose indexes exactly comprises the multiset P''. Second, it screens the ordered tuples in the product to make sure that each chosen one can be utilized to generate a proper linking mapping from P'' to the union of $\{null\}$ and the set of root elements of its constituents, and that the resulting precedence structure must be of depth $k+1$. The latter is guaranteed when one constituent of the tuple is of depth k. Third, for each chosen tuple, it generates the linking relation, which is composed of the head $\langle P'', p\rangle$, the set of tails, i.e., the root elements of the constituents of the tuple, and the newly created linking mapping, and then constructs the consequent precedence structure (E, R), where E and R comprise all the involved direct total precedence relations and relevant linking relations, respectively.

DOI reference number: 10.18293/JVLC2019N1-017

A total precedence relation is commonly represented as a precedence structure. A precedence structure can visually form a rooted tree. In Figure 4, (a), (b), and (c) show three rooted trees that correspond to three precedence structures, called ps_1, ps_2, and ps_3, respectively. Apparently, they are of depth 1, 2, and 3, respectively.

Example 2. A precedence structure $ps_3 \coloneqq (E, R)$ on the production set P of the RGG in Figure 1, where:

• $E = \{e_1, e_2, e_3, e_4\}$, in which $e_1 = \langle\{p4, p9\}, p8\rangle$, $e_2 = \langle\{p5\}, p4\rangle$, $e_3 = \langle\{p4, p8\}, p9\rangle$, $e_4 = \langle\{p2, p9\}, p8\rangle$;

• $R = \{r_1, r_2\}$, in which $r_1 = (e_1, \{e_2, e_3\}, f_1)$ with $f_1(p4) = e_2$ and $f_1(p9) = e_3$, $r_2 = (e_3, \{e_4\}, f_2)$ with $f_2(p4) = null$ and $f_2(p8) = e_4$;

• $e_t = e_1$.

In the rooted tree corresponding to ps_3, as depicted in Figure 4(c), the four fragments enclosed by red dashed rectangles are the elements e_1, e_2, e_3, and e_4, respectively, which are direct total precedence relations, i.e., precedence structures of depth 1.

The subtree enclosed by the green dashed rectangle corresponds to the precedence structure ps_2.

Given the set of total precedence relations of depth no more than 2 including ps_1 and ps_2 (the set of direct total precedence relations involved) as input, the algorithm will generate a set of total precedence relations of depth no more than 3, where ps_3 is involved. That is, ps_3 is created on the basis of ps_1 and ps_2, together with the direct total precedence relation e_1.

3.4 Computation of Context

It is known that a total precedence relation, i.e., a precedence structure (E, R), forms a rooted tree in such a way that each direct total precedence relation in E corresponds to a subtree of depth 1 and they are glued to each other in terms of the linking relations in R.

The fourth algorithm calculates all the level- i contexts of a production and respective extended productions in terms of a total precedence relation, i.e., a precedence structure, with respect to it. Readily, the diagram of a rooted tree corresponding to a precedence structure offers a more comprehensible perspective for the algorithm.

The algorithm is composed of two consecutive tasks. The first task is to transform a total precedence relation (a rooted tree) into a set of direct total precedence relations (rooted trees of depth 1).

To this end, the algorithm creates an empty set, adds the rooted tree to it, and repeats the subsequent four steps until all the elements in it become rooted trees of depth 1. First, it randomly selects a rooted tree from the set and locates an outmost subtree of length 1 (with the root node, say p); then, it figures out all the possible context- i p's according to the subtree, where $i \geq 1$; next, for each of those extended productions, say p', it constructs a new tree from the original one by pruning the subtree except the root (this node is preserved since it is also a leaf of another subtree to which this one is linked via a linking relation in R) from it and substituting p' for the root p, and puts it into the set; and finally, it deletes the originally selected tree from the set. After that, the set includes only rooted trees of depth 1, any of which corresponds to a direct total precedence relation.

Algorithm 4. Computation of contexts and extended productions.

Input. A set P of productions, a total precedence relation $\langle M, p_0\rangle = (E_0, R_0)$ of depth h.
Output. The level- h contexts of p_0 and corresponding extended productions.

{
  $Cps = \{(E_0, R_0)\}$;
  while ($\exists ps \in Cps$ such that ps is not a direct total precedence relation) {
    Let $ps = (E, R)$;
    Find an element $e \in E$ such that e is not the head of any linking relation in R;
    Let $e = \langle P', p\rangle$ and $P' = \{p_1, \cdots, p_k\}$; // $k \geq 1$;
    Let $r = (e', D, f) \in R$ such that $e \in D$;
    Let $e' = \langle P'', p''\rangle$;
    Let $Fun1[p] = h$, $Fun2[p] = l$;
    Let $l(P') = \{Q_1, \cdots, Q_m\}$; //$m \geq 1$;
    $Cps = Cps\backslash\{ps\}$;
    for each $(t_1, \cdots, t_k) \in h(Q_1) \cup \cdots \cup h(Q_m)$ {
      Construct a level- i context (U, Z), $i \geq 1$;
      Construct a context- i p with (U, Z), called p';
      $P''' = (P''\backslash\{p\}) \cup \{p'\}$;
      $e'' = \langle P''', p''\rangle$;
      $D' = D\backslash\{e\}$;
      Create a linking mapping $f': P''' \to D' \cup \{null\}$ such that $f'(p') = null$, and for any other production $p''' \in P'''$, $f'(p''') = f(p''')$;
      $r' = (e'', D', f')$;
      $E' = (E\backslash\{e, e'\}) \cup \{e''\}$;
      $R' = R\backslash\{r\} \cup \{r'\}$;
      Construct a precedence structure $ps' = (E', R')$;
      $Cps = Cps \cup \{ps'\}$;
    }
    $Cps = Cps\backslash\{ps\}$;
  }
  $Cnt = \emptyset$;
  $Xdp = \emptyset$;
  for each $ps \in Cps$ {
    Construct a level- h context (U, Z);
    $Cnt = Cnt \cup \{(U, Z)\}$;
    Construct a context- h p_0 with (U, Z), called q;
    $Xdp = Xdp \cup \{q\}$;
  }
  return (Cnt, Xdp);
}

In the second task, it constructs, for each element in the set, a level- h context and based on it, an accompanying extended production as well.

DOI reference number: 10.18293/JVLC2019N1-017

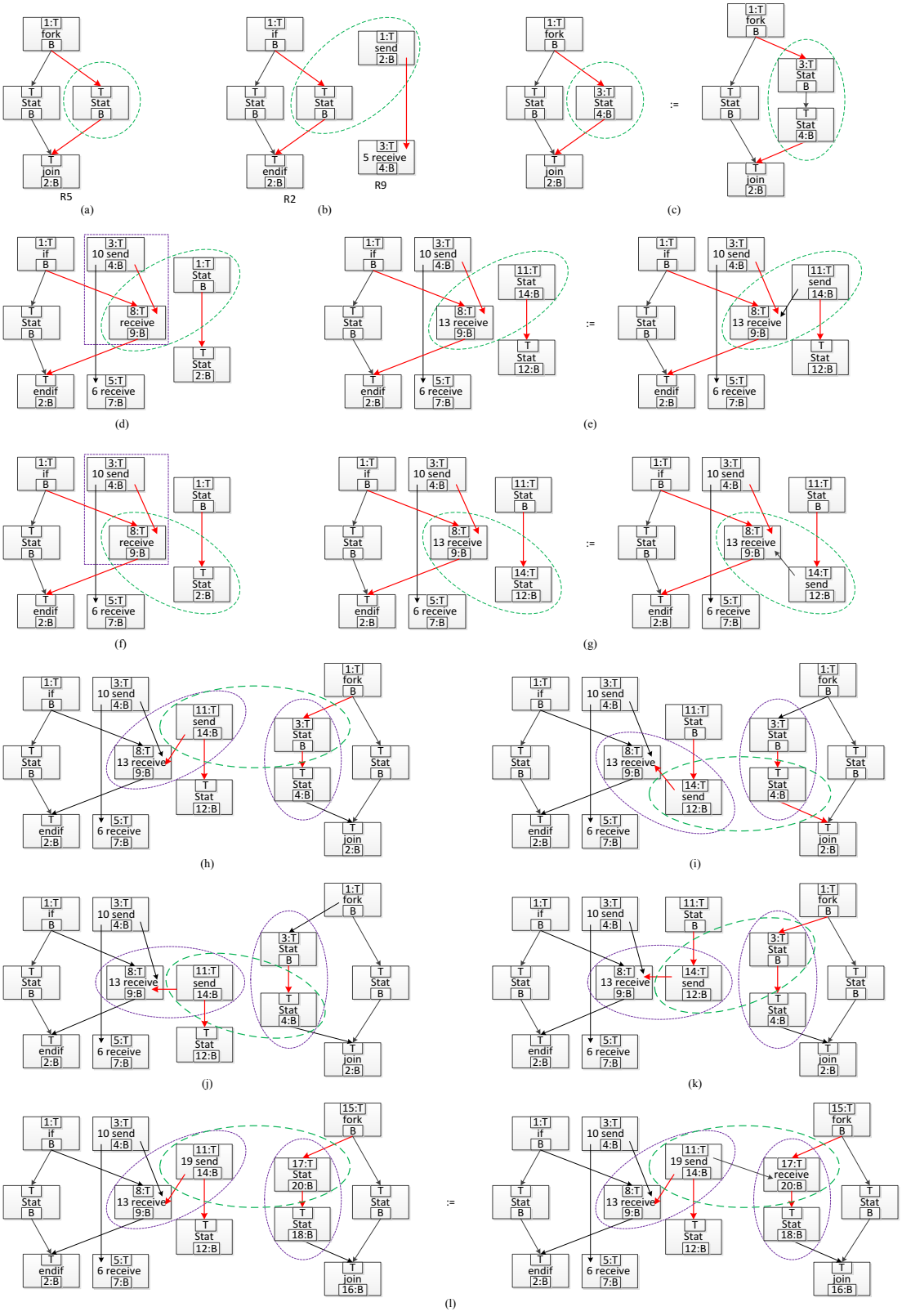

Fig. 5. Computation of Contexts. (a) A level-1 context of $p4$. (b) A level-1 context of $p8$. (c) A context-1 $p4$. (d) A level-2 context of $p9$. (e) A context-2 $p9$. (f) A level-2 context of $p9$. (g) A context-2 $p9$. (h-k) Four level-3 contexts of $p8$. (l) A context-3 $p8$.

DOI reference number: 10.18293/JVLC2019N1-017

Example 3. Computation of contexts.

Figure 5 depicts some of the contexts and context-equipped productions with respect to the RGG discussed above, which provides a visual demonstration of the computation process of Algorithm 4.

In each part of Figure 5, the subgraph enclosed by a green dashed eclipse is the left or right graph of some production, and the subgraph enclosed by a purple dashed rectangle or eclipse is the underlying production of a context-equipped production.

Figure 5(a) shows a level-1 context of $p4$, which is the output of Algorithm 4 when taking the direct total precedence relation ps_1 whose precedence structure is given in Figure 4(a) as input. The context-equipped $p4$, numbered as $p10$, is shown in (c). It is also abbreviated to context-1 $p4$.

Figure 5(b) presents a level-1 context of $p8$, with respect to the direct total precedence of e_4 whose precedence structure is shown in Figure 4(c). The corresponding context-equipped $p8$ (named by $p11$ above) is given in Figure 2(c). Taking the total precedence relation ps_2 (comprised of e_3 and e_4) shown in Figure 4(b) as input, Algorithm 4 generates two level-2 contexts of $p9$, as depicted in (d) and (f). That is, based on the right graphs of $p4$ and $p11$ (whose underlying production is $p8$), the algorithm produces all the level-2 contexts of $p9$. The quantity of contexts depends on the number of options when creating the left graph of the given production from other productions' right graphs. Accordingly, two context-equipped $p9$ are illustrated in (e) and (g), and numbered as $p12$ and $p13$, respectively.

The computation of the total precedence relation ps_3 is on the basis of ps_1 and ps_2. In a similar way, when taking the total precedence relation ps_3 as input, Algorithm 4 generates four level-3 contexts of $p8$, as shown in Figure 5(h)-(k). That is, the contexts are produced based on the right graphs of $p10$ and $p12$, or of $p10$ and $p13$. A context-equipped $p8$ that corresponds to the level-3 context in (h) is demonstrated in (l).

4. Complexity Analysis

In this section, the complexities of the proposed algorithms in the preceding section are analyzed.

Algorithm 1 calls a procedure FindRedex(H, G) to generate the set of subgraphs of H that are redexes of the marked graph G. A procedure similar to the callee was proposed in [12] with time complexity $O(|H|^{|G|})$, where $|H|$ or $|G|$ denotes the number of nodes involved in it. Then, the time complexity of the algorithm directly follows:

Proposition 1. *The time complexity of Algorithm 1 is $O(m^2n^2r^r)$, where m is the number of productions in P, n is the maximal number of components in the left or right graphs of productions in P, and r is the maximal number of nodes in any of the components.*

Proposition 2. *The time complexity of Algorithm 2 is $O((mn)^{n+1}r^{rn})$, where m, n and r are as in Proposition 1.*

Proof. The algorithm mainly consists of a for-loop that nests other three sequential for-loops.

In the outmost loop, the first nested for-loop also nests another for-loop, which takes time $O(mn^2)$. The line next to it is the calculation of a k-ary Cartesian product over sets of redexes of components in $p.L$, each of which contains at most $mn \times r^r$ elements, where mn and r^r are the maximal number of elements in Cm_R and $M[C, C']$, respectively. Thus, the time complexity is $O((mnr^r)^n)$, since k equals to n in the worst case.

In the second nested for-loop, the first line takes $O(n)$, because the production to which each redex in the ordered tuple of Rlt corresponds is clearly indicated beforehand, according to the underlying assumption. Therefore, this for-loop takes time $O(n(mnr^r)^n)$.

Moreover, the time complexity of the last nested for-loop is at most $O(n(mnr^r)^n)$, for the cardinality of $Dtp[p]$ is no more than that of $Rlt[p]$.

In summary, the nested part of the outmost loop takes time $O(n(mnr^r)^n)$. Consequently, the time complexity of the whole procedure is $O((mn)^{n+1}r^{rn})$, which is the product of the time complexity of the outmost for-loop and that of its nested part. ■

Proposition 3. *The time complexity of Algorithm 3 is $O(m^{n+2}nl^n)$, where m and n are as in Proposition 1, and l is the cardinality of Tpd.*

Proof. The algorithm consists of three sequential for-loops. It is evident that the first two loops take $O(n)$ and $O(l)$, respectively.

The last one is a four-layer nested for-loops. According to the structure, its time complexity can be expressed as $O(d_1d_2d_3d_4)$, where d_1, d_2, d_3 and d_4 are the maximal number of times traversed in the outmost, second, third, and inmost for-loop, respectively. We proceed inwards from outside of the structure.

First, d_1 is the number of productions in P, that is, $d_1 = m$. Next, d_2 equals to the cardinality of $\preccurlyeq_P$, which is no more than $m \times m^n = m^{n+1}$. It is obvious that d_3 is actually the cardinality of the Cartesian product over k sets from Ptp, where k denotes the number of tails in a direct total precedence relation. Since Ptp is a partition of

DOI reference number: 10.18293/JVLC2019N1-017

Tpd, each element of the former must be a subset of the latter. Thus, $d_3 < l^n$. As for the inmost loop, d_4 readily equals to n, which is exactly the same as the exponent appearing in the preceding inequation.

Consequently, the last structure takes $O(m^{n+2}nl^n)$, i.e., the product of the complexities of the four constituents. Readily, it is also the time complexity of the algorithm. ■

Theorem 1. *The time complexity of Algorithm 4 is* $O((n!\,r^{rn})^{2n^{h-1}})$*, where* n *and* r *are as in Proposition 1, and* h *is the depth of the input total precedence relation.*

Proof. The main part of the algorithm is a two-layer structure: a for-loop nested within a while-loop, followed by another for-loop.

As to the former, the total number of times it is traversed is the product of that of the outmost while-loop and of the inmost for-loop. We consider the while-loop first. In the worst case, the input total precedence relation (E_0, R_0) corresponds to a complete n-ary rooted tree of depth h. Then, the cardinality of E_0, i.e., the number of subtrees of depth 1 that compose it, can be expressed as:

$$1 + n + \cdots + n^{h-1} = \frac{n^h - 1}{n-1} \tag{1}$$

Suppose the number of times the inmost for-loop is traversed at the worst case be w. In each traversal of the while-loop, it takes a rooted tree out from Cps, and then puts as many as w revised ones that comprise one less subtrees back into it. This process is repeated until each element in Cps becomes a rooted tree of depth 1, i.e., it only involves a root element. Consequently, the maximal number of times the loop is traversed can be expressed as:

$$1 + w + w^2 \cdots + w^{(\frac{n^h-1}{n-1}-1)} = \frac{w^{\frac{n^h-1}{n-1}} - 1}{w-1} \tag{2}$$

Further inference to Formula (2) can be done as follows:

$$\frac{w^{\frac{n^h-1}{n-1}} - 1}{w-1} < \frac{w}{w-1} \cdot w^{\frac{n^h-1}{n-1}-1} = \frac{w}{w-1} \cdot w^{\frac{n(n^{h-1}-1)}{n-1}}$$
$$< \frac{w}{w-1} \cdot w^{2n^{h-1}} = O(w^{2n^{h-1}})$$

Note that w exactly equals to the number of extended context-1 productions that can be produced from a direct total precedence relation, by Definition 5. In the worst case, the relation consists of one head and n tails, and each component of the latter's right graphs contains r^r redexes of any component of the former's left graph. To be exact, of the loop variable $(t_1, \cdots, t_k)$ for the for-loop, each element t_i can be any of the r^r redexes of the corresponding component of the left graph in any component of any tail's right graph, where $1 \le i \le k$, and $k = n$. Thus, for each permutation of the tails, there are $(r^r)^n = r^{rn}$ possible ordered tuples, where n is the maximal number of choices for selecting one component from each tail, and r^r is the maximal number of redexes in each component. Furthermore, the number of permutations for the tails is $n!$. Therefore, $w = n!\,r^{rn}$. Substituting the result for w in Formula (2) yields $(n!\,r^{rn})^{2n^{h-1}}$.

As to the latter, the number of times it is traversed is $n^{h-1} \times n = n^h$.

Consequently, the time complexity of the algorithm is $O((n!\,r^{rn})^{2n^{h-1}})$. ■

5. Discussion

5.1 Applicability of the Algorithms

From the perspective of time complexity, the above four algorithms seem rather complicated at first glance. Nevertheless, they are applicable in practical scenarios due to the following three causes.

First, the parameters that characterize a graph grammar are usually small constants, and cannot change in any computation. That is, the parameters are the nature of a graph grammar that will not vary with host graphs in parsing or derivation processes under different situations.

Second, the worst cases theoretically assumed in the analysis of the complexities can rarely happen in practice, and they are frequently quite small number in practical applications. Notice that the number of redexes with respect to a direct total precedence relation is surprisingly $n!\,r^{rn}$. However, this amount is merely a theoretical upper bound that accounts for all the possibly matched subgraphs, no matter in which situation all the redexes can simultaneously occur. An extreme situation in this case is a host graph where all the nodes in the host graph are labeled with the same symbol and the directed edges between them are completely connected. However, such host graphs can rarely be encountered in practice. As an example, consider the graph grammar depicted in Figure 1, the number of redexes with respect to a direct total precedence relation is theoretically $2!\,4^{2\times4}$, whereas in the practical computation it is less than 10 in most cases.

Third, the contexts of a graph grammar can be achieved as the output from merely one execution of the algorithms, and then they can be repeatedly utilized in the process of derivation and parsing of this grammar at any time afterwards.

5.2 Application of Context

Context offers a concrete way for designers or users to grasp the meaning of an implicit graph grammar by directly observing the productions instead of enumerating

DOI reference number: 10.18293/JVLC2019N1-017

the members of the language. A context of a production characterizes a potential circumstance, under which it can be applied for derivation, a means usually employed to generate members of the language. Conversely, the context can also be regarded as a circumstance under which the production can be applied for parsing. Any production is self-explanatory for what it is for, whereas the contexts at distinct levels indicate at which situations it can be applied. These two aspects together clearly show the intension or meaning of a production, from the point view of derivation. Consequently, contexts can facilitate the comprehension of a graph grammar by synthesizing the meanings of its constituents so as to constitute the overall characteristics of the members of its language.

Moreover, context can facilitate the improvement of parsing performance. A general parsing algorithm is always a necessity for graph grammar formalisms. Backtracking is the main cause of high time complexity of a general parsing algorithm. In the process of parsing a host graph, when some unexpected (false positive) redexes are found and the corresponding reductions are conducted accordingly, then a final graph may be obtained that is not the initial graph of the involved graph grammar and to which no more reductions can be done. This situation gives rise to backtracking. A redex is called false positive if the situation in which the redex lies does not match any of the contexts of the production. Consequently, identifying the false positive redexes so as to avoid unexpected reductions is an effective way to improve parsing performance. Apparently, context matching can serve this purpose.

Noticeably, the proposed approach to context computation can be directly applied to practical graph grammars specifying real-world visual languages, e.g., BPMN (Business Process Model and Notation), ER diagrams, UML diagrams, WebML (Web Modeling Language), chemical diagrams, and so on, since these graph grammars are concrete examples of the underlying formalisms where the specification of nodes and edges in productions is entirely inherited from the formalisms.

6. Conclusion

On the basis of RGG, a representative of implicit context-sensitive graph grammar formalism, this paper has proposed an approach to the computation of context according to the formal definition of context, and presented the time complexities of the partially ordered algorithms involved. The method can facilitate the applicability of implicit graph grammars, as contexts of the productions are essential information for the comprehension of graph grammars and the improvement of parsing performance of parsing algorithms. Besides, the method can be generalized to other implicit context-sensitive graph grammar formalisms without much effort.

In the future, further investigation will be conducted to explore more application scenarios of context, and a support system for context computation and visualization in a context-sensitive graph grammar framework will be developed as well.

Acknowledgments

This work is supported in part by the National Science Foundation of China under grants 61170089 and 91318301.

References

[1] Chang S. K. (1987) “Visual Languages: A Tutorial and Survey”. IEEE Software, 4(1), pp. 29–39.

[2] Marriott K. (1994) “Constraint Multiset Grammars”. IEEE Symposium on Visual Languages, St. Louis, Missouri, pp. 118–125.

[3] Ferrucci F., Pacini G., Satta G., et al. (1996) “Symbol-Relation Grammars: A Formalism For Graphical Languages”. Information and Computation, 131(1), pp. 1–46.

[4] Chang S. K. (1971) “Picture Processing Grammar and Its Applications”. Information Sciences, Vol. 3, pp.121–148.

[5] Lakin F. (1987) “Visual Grammars for Visual Languages”. 7th National Conference on Artificial Intelligence, pp. 683–688.

[6] You K. C., Fu K. S. (1979) “A Syntactic Approach to Shape Recognition Using Attributed Grammars”. IEEE Transactions on Systems, Man and Cybernetics, 9(6), pp. 334–345.

[7] Costagliola G., Deufemia V., Polese G. (2007) “Visual Language Implementation Through Standard Compiler-Compiler Techniques”. Journal of Visual Languages and Computing, 18(2), pp. 165-226.

[8] Rozenberg G. (Ed.) (1997) “Handbook on Graph Grammars and Computing by Graph Transformation, Vol.1: Foundations”. World Scientific.

[9] Engels G., Kreowski H. J., Rozenberg G. (Eds.) (1999) “Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages, and Tools”. World Scientific.

[10] Ehrig H., Kreowski H. J., Montanari U., Rozenberg G. (Eds.) (1999) “Handbook of Graph Grammars and Computing by Graph Transformation, Vol.3: Concurrency, Parallelism, and Distribution”. World Scientific.

[11] Rekers J., Schürr A. (1997) “Defining and Parsing Visual Languages with Layered Graph Grammars”. Journal of Visual Languages and Computing, 8(1), pp. 27–55.

[12] Zhang D., Zhang K., Cao J. (2001) “A Context-Sensitive Graph Grammar Formalism for the Specification of Visual Languages”. The Computer Journal, 44(3), pp.187–200.

[13] Nagl M. (1979) “A Tutorial and Bibliographical Survey on Graph Grammars”. International Workshop on Graph Grammars and Their Application to Computer Science and Biology, Lecture Notes in Computer Science, Vol. 73, Springer Verlag, pp. 70–126.

[14] Nagl M. (1987) “Set Theoretic Approaches to Graph Grammars”. International Workshop on Graph Grammars and Their Application to Computer Science, Lecture Notes in Computer Science, Vol. 291, Springer Verlag, pp. 41–54.

[15] Shi Z., Zeng X., Zou Y., et al. (2018) “A Temporal Graph Grammar Formalism,” Journal of Visual Languages and Computing, Vol. 47, pp. 62–76.

DOI reference number: 10.18293/JVLC2019N1-017

[16] Kong J., Zhang K., Zeng X. (2006) "Spatial Graph Grammars for Graphical User Interfaces". ACM Transactions on Computer-Human Interaction, 13(2), pp. 268–307.

[17] Zhao C., Kong J., Zhang K. (2010) "Program Behavior Discovery and Verification: A Graph Grammar Approach". IEEE Transactions on Software Engineering, 36(3), pp. 431–448.

[18] Roudaki A., Kong J., Zhang K. (2016) "Specification and Discovery of Web Patterns: A Graph Grammar Approach". Information Sciences, Vol. 328, 528-545.

[19] Liu Y., Zeng X., Zou Y., Zhang K. (2018) "A Graph Grammar-Based Approach for Graph Layout," Software: Practice and Experience, 49(8), pp. 1523–1535.

[20] Kong J., Barkol O., Bergman R., Pnueli A., Schein S., et al. (2012) "Web Interface Interpretation Using Graph Grammars". IEEE Transactions on System, Man, and Cybernetics – Part C, 42(4), pp. 590–602.

[21] Chen L., Huang L., Chen L. (2015) "Breeze Graph grammar: A Graph Grammar Approach for Modeling the Software Architecture of Big Data-Oriented Software Systems". Software: Practice and Experience, 45(8), pp. 1023–1050.

[22] Liu Y., Zhang K., Kong J., Zou Y., Zeng X. (2018) "Spatial Specification and Reasoning Using Grammars: From Theory to Application," Spatial Cognition & Computation, Taylor & Francis, 18(4), pp. 315–340.

[23] Pfaltz J. L., Rosefeld A. (1969) "Web Grammars". International Joint Conference on Artificial Intelligence, pp. 609–619.

[24] Zeng X., Han X., Zou Y. (2008) "An Edge-Based Context-Sensitive Graph Grammar Formalism". Journal of Software, 19(8), pp. 1893–1901. (in Chinese)

[25] Liu Y., Shi Z., Wang Y. (2018) "An Edge-Based Graph Grammar Formalism and Its Support System". International DMS Conference on Visualization and Visual Languages, pp.101–108.

[26] Zou Y., Zeng X., Han X. (2008) "Context-Attributed Graph Grammar Framework for Specifying Visual Languages". Journal of Southeast University (English Edition), 24(4), pp. 455–461.

[27] Bottoni P., Taentzer G., Schürr A. (2000) "Efficient Parsing of Visual Languages Based on Critical Pair Analysis and Contextual Layered Graph Transformation". IEEE Symposium on Visual Languages, pp. 59–60.

[28] Zou Y, Lü J, Tao X. (2019) "Research on Context of Implicit Context-Sensitive Graph Grammars". Journal of Computer Languages, Vol. 51, pp. 241–260.

DOI reference number: 10.18293/JVLC2019N1-017