HAL
open science

Parsing Directed Acyclic Graphs with Range Concatenation Grammars

Pierre Boullier, Benoît Sagot

To cite this version:

Pierre Boullier, Benoît Sagot. Parsing Directed Acyclic Graphs with Range Concatenation Grammars. International Conference on Parsing Technologies (IWPT 2009), 2009, Paris, France. inria-00616690

HAL Id: inria-00616690

https://inria.hal.science/inria-00616690v1

Submitted on 23 Aug 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Parsing Directed Acyclic Graphs with Range Concatenation Grammars

Pierre Boullier and Benoît Sagot
Alpage, INRIA Paris-Rocquencourt & Université Paris 7
Domaine de Voluceau — Rocquencourt, BP 105 — 78153 Le Chesnay Cedex, France
{Pierre.Boullier,Benoit.Sagot}@inria.fr

Abstract

Range Concatenation Grammars (RCGs) are a syntactic formalism which possesses many attractive properties. It is more powerful than Linear Context-Free Rewriting Systems, though this power is not reached to the detriment of efficiency since its sentences can always be parsed in polynomial time. If the input, instead of a string, is a Directed Acyclic Graph (DAG), only *simple* RCGs can still be parsed in polynomial time. For non-linear RCGs, this polynomial parsing time cannot be guaranteed anymore. In this paper, we show how the standard parsing algorithm can be adapted for parsing DAGs with RCGs, both in the linear (simple) and in the non-linear case.

1 Introduction

The Range Concatenation Grammar (RCG) formalism has been introduced by Boullier ten years ago. A complete definition can be found in (Boullier, 2004), together with some of its formal properties and a parsing algorithm (qualified here of *standard*) which runs in polynomial time. In this paper we shall only consider the positive version of RCGs which will be abbreviated as PRCG.[1] PRCGs are very attractive since they are more powerful than the Linear Context-Free Rewriting Systems (LCFRSs) by (Vijay-Shanker et al., 1987). In fact LCFRSs are equivalent to simple PRCGs which are a subclass of PRCGs. Many Mildly Context-Sensitive (MCS) formalisms, including Tree Adjoining Grammars (TAGs) and various kinds of Multi-Component TAGs, have already been translated into their simple PRCG counterpart in order to get an efficient parser for free (see for example (Barthélemy et al., 2001)).

However, in many Natural Language Processing applications, the most suitable input for a parser is not a sequence of words (forms, terminal symbols), but a more complex representation, usually defined as a Direct Acyclic Graph (DAG), which correspond to finite regular languages, for taking into account various kinds of ambiguities. Such ambiguities may come, among others, from the output of speech recognition systems, from lexical ambiguities (and in particular from tokenization ambiguities), or from a non-deterministic spelling correction module.

Yet, it has been shown by (Bertsch and Nederhof, 2001) that parsing of regular languages (and therefore of DAGs) using simple PRCGs is polynomial. In the same paper, it is also proven that parsing of finite regular languages (the DAG case) using arbitrary RCGs is NP-complete.

This papers aims at showing how these complexity results can be made concrete in a parser, by extending a standard RCG parsing algorithm so as to handle input DAGs. We will first recall both some basic definitions and their notations. Afterwards we will see, with a slight modification of the notion of ranges, how it is possible to use the standard PRCG parsing algorithm to get in polynomial time a parse forest with a DAG as input.[2] However, the resulting parse forest is valid only for simple PRCGs. In the non-linear case, and consistently with the complexity results mentioned above, we show that the resulting parse forest needs further processing for filtering out inconsistent parses, which may need an exponential time. The proposed filtering algorithm allows for parsing DAGs in practice with any PRCG, including non-linear ones.

[1] Negative RCGs do not add formal power since both versions exactly cover the class *PTIME* of languages recognizable in deterministic polynomial time (see (Boullier, 2004) for an indirect proof and (Bertsch and Nederhof, 2001) for a direct proof).

[2] The notion of parse forest is reminiscent of the work of (Lang, 1994).

2 Basic notions and notations

2.1 Positive Range Concatenation Grammars

A *positive range concatenation grammar* (PRCG) $G = (N, T, V, P, S)$ is a 5-tuple in which:

- T and V are disjoint alphabets of *terminal symbols* and *variable symbols* respectively.
- N is a non-empty finite set of *predicates* of fixed *arity* (also called *fan-out*). We write $k = \mathit{arity}(A)$ if the arity of the predicate A is k. A predicate A with its *arguments* is noted $A(\vec{\alpha})$ with a vector notation such that $|\vec{\alpha}| = k$ and $\vec{\alpha}[j]$ is its jth argument. An argument is a string in $(V \cup T)^*$.
- S is a distinguished predicate called the *start predicate* (or *axiom*) of arity 1.
- P is a finite set of *clauses*. A clause c is a rewriting rule of the form $A_0(\vec{\alpha_0}) \rightarrow A_1(\vec{\alpha_1}) \dots A_r(\vec{\alpha_r})$ where r, $r \geq 0$ is its *rank*, $A_0(\vec{\alpha_0})$ is its *left-hand side* or *LHS*, and $A_1(\vec{\alpha_1}) \dots A_r(\vec{\alpha_r})$ its *right-hand side* or *RHS*. By definition $c[i] = A_i(\vec{\alpha_i})$, $0 \leq i \leq r$ where A_i is a predicate and $\vec{\alpha_i}$ its arguments; we note $c[i][j]$ its jth argument; $c[i][j]$ is of the form $X_1 \dots X_{n_{ij}}$ (the X_k's are terminal or variable symbols), while $c[i][j][k]$, $0 \leq k \leq n_{ij}$ is a *position* within $c[i][j]$.

For a given clause c, and one of its predicates $c[i]$ a *subargument* is defined as a substring of an argument $c[i][j]$ of the predicate $c[i]$. It is denoted by a pair of positions $(c[i][j][k], c[i][j][k'])$, with $k \leq k'$.

Let $w = a_1 \dots a_n$ be an input string in T^*, each occurrence of a substring $a_{l+1} \dots a_u$ is a pair of positions $(w[l], w[u])$ s.t. $0 \leq l \leq u \leq n$ called a *range* and noted $\langle l..u \rangle_w$ or $\langle l..u \rangle$ when w is implicit. In the range $\langle l..u \rangle$, l is its *lower bound* while u is its *upper bound*. If $l = u$, the range $\langle l..u \rangle$ is an *empty* range, it spans an empty substring. If $\rho_1 = \langle l_1..u_1 \rangle$, $\dots$ and $\rho_m = \langle l_m..u_m \rangle$ are ranges, the *concatenation* of $\rho_1, \dots, \rho_m$ noted $\rho_1 \dots \rho_m$ is the range $\rho = \langle l..u \rangle$ if and only if we have $u_i = l_{i+1}$, $1 \leq i < m$, $l = l_1$ and $u = u_m$.

If $c = A_0(\vec{\alpha_0}) \rightarrow A_1(\vec{\alpha_1}) \dots A_r(\vec{\alpha_r})$ is a clause, each of its subarguments $(c[i][j][k], c[i][j][k'])$ may take a range $\rho = \langle l..u \rangle$ as value: we say that it is *instantiated* by ρ. However, the instantiation of a subargument is subjected to the following constraints.

- If the subargument is the empty string (i.e., $k = k'$), ρ is an empty range.
- If the subargument is a terminal symbol (i.e., $k + 1 = k'$ and $X_{k'} \in T$), ρ is such that $l + 1 = u$ and $a_u = X_{k'}$. Note that several occurrences of the same terminal symbol may be instantiated by different ranges.
- If the subargument is a variable symbol (i.e., $k + 1 = k'$ and $X_{k'} \in V$), any occurrence $(c[i'][j'][m], c[i'][j'][m'])$ of $X_{k'}$ is instantiated by ρ. Thus, each occurrence of the same variable symbol must be instantiated by the same range.
- If the subargument is the string $X_{k+1} \dots X_{k'}$, ρ is its instantiation if and only if we have $\rho = \rho_{k+1} \dots \rho_{k'}$ in which ρ_{k+1}, $\dots$, $\rho_{k'}$ are respectively the instantiations of X_{k+1}, $\dots$, $X_{k'}$.

If in c we replace each argument by its instantiation, we get an *instantiated clause* noted $A_0(\vec{\rho_0}) \rightarrow A_1(\vec{\rho_1}) \dots A_r(\vec{\rho_r})$ in which each $A_i(\vec{\rho_i})$ is an *instantiated predicate*.

A binary relation called *derive* and noted $\underset{G,w}{\Rightarrow}$ is defined on strings of instantiated predicates. If Γ_1 and Γ_2 are strings of instantiated predicates, we have

$$\Gamma_1\ A_0(\vec{\rho_0})\ \Gamma_2 \underset{G,w}{\Rightarrow} \Gamma_1\ A_1(\vec{\rho_1}) \dots A_m(\vec{\rho_m})\ \Gamma_2$$

if and only if $A_0(\vec{\rho_0}) \rightarrow A_1(\vec{\rho_1}) \dots\ A_m(\vec{\rho_m})$ is an instantiated clause.

The *(string) language* of a PRCG G is the set $\mathcal{L}(G) = \{w \mid S(\langle 0..|w| \rangle_w) \overset{+}{\underset{G,w}{\Rightarrow}} \varepsilon\}$. In other words, an input string $w \in T^*$, $|w| = n$ is a *sentence* of G if and only there exists a *complete derivation* which starts from $S(\langle 0..n \rangle)$ (the instantiation of the start predicate on the whole input text) and leads to the empty string (of instantiated predicates). The *parse forest* of w is the CFG whose axiom is $S(\langle 0..n \rangle)$ and whose productions are the instantiated clauses used in all complete derivations.[3]

We say that the arity of a PRCG is k, and we call it a k-PRCG, if and only if k is the maximum

[3]Note that this parse forest has no terminal symbols (its language is the empty string).

arity of its predicates ($k = \max_{A \in N} arity(A)$). We say that a k-PRCG is *simple*, we have a simple k-PRCG, if and only if each of its clause is

- *non-combinatorial*: the arguments of its RHS predicates are single variables;
- *non-erasing*: each variable which occur in its LHS (resp. RHS) also occurs in its RHS (resp. LHS);
- *linear*: there are no variables which occur more than once in its LHS and in its RHS.

The subclass of simple PRCGs is of importance since it is MCS and is the one equivalent to LCFRSs.

2.2 Finite Automata

A *non-deterministic finite automaton* (NFA) is the 5-tuple $A = (Q, \Sigma, \delta, q_0, F)$ where Q is a non empty finite set of *states*, Σ is a finite set of *terminal symbols*, δ is the ternary *transition relation* $\delta = \{(q_i, t, q_j) | q_i, q_j \in Q \wedge t \in \Sigma \cup \{\varepsilon\}\}$, q_0 is a distinguished element of Q called the *initial state* and F is a subset of Q whose elements are called *final states*. The size of A, noted $|A|$, is its number of states ($|A| = |Q|$).

We define the ternary relation δ^* on $Q \times \Sigma^* \times Q$ as the smallest set s.t. $\delta^* = \{(q, \varepsilon, q) \mid q \in Q\} \cup \{(q_1, xt, q_3) \mid (q_1, x, q_2) \in \delta^* \wedge (q_2, t, q_3) \in \delta\}$. If $(q, x, q') \in \delta^*$, we say that x is a *path* between q and q'. If $q = q_0$ and $q' \in F$, x is a *complete path*.

The *language* $\mathcal{L}(A)$ *defined* (*generated*, *recognized*, *accepted*) by the NFA A is the set of all its complete paths.

We say that a NFA is *empty* if and only if its language is empty. Two NFAs are *equivalent* if and only if they define the same language. A NFA is *ε-free* if and only if its transition relation does not contain a transition of the form (q_1, ε, q_2). Every NFA can be transformed into an equivalent ε-free NFA (this classical result and those recalled below can be found, e.g., in (Hopcroft and Ullman, 1979)).

As usual, a NFA is drawn with the following conventions: a transition (q_1, t, q_2) is an arrow labelled t from state q_1 to state q_2 which are printed with a surrounded circle. Final states are doubly circled while the initial state has a single unconnected, unlabelled input arrow.

A *deterministic finite automaton* (DFA) is a NFA in which the transition relation δ is a *transition function*, $\delta : Q \times \Sigma \rightarrow Q$. In other words, there are no ε-transitions and if $(q_1, t, q_2) \in \delta$, $t \neq \varepsilon$ and $\nexists (q_1, t, q_2') \in \delta$ with $q_2' \neq q_2$. Each NFA can be transformed by the *subset construction* into an equivalent DFA. Moreover, each DFA can be transformed by a *minimization algorithm* into an equivalent DFA which is *minimal* (i.e., there is no other equivalent DFA with fewer states).

2.3 Directed acyclic graphs

Formally, a directed acyclic graph (DAG) $D = (Q, \Sigma, \delta, q_0, F)$ is an NFA for which there exists a strict order relation $<$ on Q such that $(p, t, q) \in \delta \Rightarrow p < q$. Without loss of generality we may assume that $<$ is a total order.

Of course, as NFAs, DAGs can be transformed into equivalent deterministic or minimal DAGs.

3 DAGs and PRCGs

A DAG D is *recognized* (*accepted*) by a PRCG G if and only if $\mathcal{L}(D) \cap \mathcal{L}(G) \neq \emptyset$. A trivial way to solve this recognition (or parsing) problem is to extract the complete paths of $\mathcal{L}(D)$ (which are in finite number) one by one and to parse each such string with a standard PRCG parser, the (complete) parse forest for D being the union of each individual forest.[4] However since DAGs may define an exponential number of strings w.r.t. its own size,[5] the previous operation would take an exponential time in the size of D, and the parse forest would also have an exponential size.

The purpose of this paper is to show that it is possible to directly parse a DAG (without any unfolding) by sharing identical computations. This sharing may lead to a polynomial parse time for an exponential number of sentences, but, in some cases, the parse time remains exponential.

3.1 DAGs and Ranges

In many NLP applications the source text cannot be considered as a sequence of terminal symbols, but rather as a finite set of finite strings. As

[4] These forests do not share any production (instantiated clause) since ranges in a particular forest are all related to the corresponding source string w (i.e., are all of the form $\langle i..j \rangle_w$). To be more precise the union operation on individual forests must be completed in adding productions which connect the new (super) axiom (say S') with each root and which are, for each w of the form $S' \rightarrow S(\langle 0..|w| \rangle_w)$.

[5] For example the language $(a|b)^n$, $n > 0$ which contains 2^n strings can be defined by a minimal DAG whose size is $n + 1$.

mentioned in th introduction, this non-unique string could be used to encode not-yet-solved ambiguities in the input. DAGs are a convenient way to represent these finite sets of strings by factorizing their common parts (thanks to the minimization algorithm).

In order to use DAGs as inputs for PRCG parsing we will perform two generalizations.

The first one follows. Let $w = t_1 \dots t_n$ be a string in some alphabet Σ and let $Q = \{q_i \mid 0 \leq i \leq n\}$ be a set of $n+1$ *bounds* with a total order relation $<$, we have $q_0 < q_1 < \dots < q_n$. The sequence $\pi = q_0 t_1 q_1 t_2 q_2 \dots t_n q_n \in Q \times (\Sigma \times Q)^n$ is called a *bounded string* which *spells* w. A range is a pair of bounds (q_i, q_j) with $q_i < q_j$ noted $\langle p_i..p_j \rangle_\pi$ and any triple of the form $(q_{i-1} t_i q_i)$ is called a *transition*. All the notions around PRCGs defined in Section 2.1 easily generalize from strings to bounded strings. It is also the case for the standard parsing algorithm of (Boullier, 2004).

Now the next step is to move from bounded strings to DAGs. Let $D = (Q, \Sigma, \delta, q_0, F)$ be a DAG. A string $x \in \Sigma^*$ s.t. we have $(q_1, x, q_2) \in \delta^*$ is called a *path* between q_1 and q_2 and a string $\pi = q t_1 q_1 \dots t_p q_p \in Q \times (\Sigma \cup \{\varepsilon\} \times Q)^*$ is a *bounded path* and we say that π *spells* $t_1 t_2 \dots t_p$. A path x from q_0 to $f \in F$ is a *complete path* and a bounded path of the form $q_0 t_1 \dots t_n f$ with $f \in F$ is a *complete bounded path*. In the context of a DAG D, a range is a pair of states (q_i, q_j) with $q_i < q_j$ noted $\langle q_i..q_j \rangle_D$. A range $\langle q_i..q_j \rangle_D$ is *valid* if and only if there exists a path from q_i to q_j in D. Of course, any range $\langle p..q \rangle_D$ defines its associated sub-DAG $D_{\langle p..q \rangle} = (Q_{\langle p..q \rangle}, \Sigma_{\langle p..q \rangle}, \delta_{\langle p..q \rangle}, p, \{q\})$ as follows. Its transition relation is $\delta_{\langle p..q \rangle} = \{(r, t, s) \mid (r, t, s) \in \delta \wedge (p, x', r), (s, x'', q) \in \delta^*\}$. If $\delta_{\langle p..q \rangle} = \emptyset$ (i.e., there is no path between p and q), $D_{\langle p..q \rangle}$ is the empty DAG, otherwise $Q_{\langle p..q \rangle}$ (resp. $\Sigma_{\langle p..q \rangle}$) are the states (resp. terminal symbols) of the transitions of $\delta_{\langle p..q \rangle}$. With this new definition of ranges, the notions of instantiation and derivation easily generalize from bounded strings to DAGs.

The language of a PRCG G for a DAG D is defined by $\overset{\bullet}{\mathcal{L}}(G, D) = \bigcup_{f \in F} \{x \mid S(\langle q_0..f \rangle_D) \overset{+}{\underset{G,D}{\Rightarrow}} \varepsilon\}$. Let $x \in \mathcal{L}(D)$, it is not very difficult to show that if $x \in \mathcal{L}(G)$ then we have $x \in \overset{\bullet}{\mathcal{L}}(G, D)$. However, the converse is not true (see Example 1), a sentence of $\mathcal{L}(D) \cap \overset{\bullet}{\mathcal{L}}(G, D)$ may not be in $\mathcal{L}(G)$. To put it differently, if we use the standard RCG parser, with the ranges of a DAG, we produce the shared parse-forest for the language $\overset{\bullet}{\mathcal{L}}(G, D)$ which is a superset of $\mathcal{L}(D) \cap \mathcal{L}(G)$.

However, if G is a simple PRCG, we have the equality $\mathcal{L}(G) = \bigcup_{D \text{ is a DAG}} \overset{\bullet}{\mathcal{L}}(G, D)$. Note that the subclass of simple PRCGs is of importance since it is MCS and it is the one equivalent to LCFRSs. The informal reason of the equality is the following. If an instantiated predicate $A_i(\vec{\rho_i})$ succeeds in some RHS, this means that each of its ranges $\vec{\rho_i}[j] = \langle k..l \rangle_D$ has been recognized as being a component of A_i, more precisely their exists a path from k to l in D which is a component of A_i. The range $\langle k..l \rangle_D$ selects in D a set $\delta_{\langle k..l \rangle_D}$ of transitions (the transitions used in the bounded paths from k to l). Because of the linearity of G, there is no other range in that RHS which selects a transition in $\delta_{\langle k..l \rangle_D}$. Thus the bounded paths selected by all the ranges of that RHS are disjoints. In other words, any occurrence of a valid instantiated range $\langle i..j \rangle_D$ selects a set of paths which is a subset of $\mathcal{L}(D_{\langle i..j \rangle})$.

Now, if we consider a non-linear PRCG, in some of its clauses, there is a variable, say X, which has several occurrences in its RHS (if we consider a top-down non-linearity). Now assume that for some input DAG D, an instantiation of that clause is a component of some complete derivation. Let $\langle p..q \rangle_D$ be the instantiation of X in that instantiated clause. The fact that a predicate in which X occurs succeeds means that there exist paths from p to q in $D_{\langle p..q \rangle}$. The same thing stands for all the other occurrences of X but nothing force these paths to be identical or not.

Example 1.

Let us take an example which will be used throughout the paper. It is a non-linear 1-PRCG which defines the language $a^n b^n c^n$, $n \geq 0$ as the intersection of the two languages $a^* b^n c^n$ and $a^n b^n c^*$. Each of these languages is respectively defined by the predicates $a^* b^n c^n$ and $a^n b^n c^*$; the start predicate is $a^n b^n c^n$.

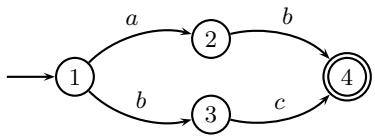

Figure 1: Input DAG associated with $ab|bc$.

$$
\begin{array}{lcl}
a^n b^n c^n(X) & \rightarrow & a^* b^n c^n(X)\ a^n b^n c^*(X) \\
\\
a^* b^n c^n(aX) & \rightarrow & a^* b^n c^n(X) \\
a^* b^n c^n(X) & \rightarrow & b^n c^n(X) \\
b^n c^n(bXc) & \rightarrow & b^n c^n(X) \\
b^n c^n(\varepsilon) & \rightarrow & \varepsilon \\
\\
a^n b^n c^*(Xc) & \rightarrow & a^n b^n c^*(X) \\
a^n b^n c^*(X) & \rightarrow & a^n b^n(X) \\
a^n b^n(aXb) & \rightarrow & a^n b^n(X) \\
a^n b^n(\varepsilon) & \rightarrow & \varepsilon
\end{array}
$$

If we use this PRCG to parse the DAG of Figure 1 which defines the language $\{ab, bc\}$, we (erroneously) get the non-empty parse forest of Figure 2 though neither ab nor bc is in $a^n b^n c^n$.[6] *It is not difficult to see that the problem comes from the non-linear instantiated variable $X_{\langle 1..4\rangle}$ in the start node, and more precisely from the actual (wrong) meaning of the three different occurrences of $X_{\langle 1..4\rangle}$ in $a^n b^n c^n(X_{\langle 1..4\rangle}) \rightarrow a^* b^n c^n(X_{\langle 1..4\rangle})\ a^n b^n c^*(X_{\langle 1..4\rangle})$. The first occurrence in its RHS says that there exists a path in the input DAG from state 1 to state 4 which is an $a^* b^n c^n$. The second occurrence says that there exists a path from state 1 to state 4 which is an $a^n b^n c^*$. While the LHS occurrence (wrongly) says that there exists a path from state 1 to state 4 which is an $a^n b^n c^n$. However, if the two $X_{\langle 1..4\rangle}$'s in the RHS had selected common paths (this is not possible here) between 1 and 4, a valid interpretation could have been proposed.*

With this example, we see that the difficulty of DAG parsing only arises with non-linear PRCGs.

If we consider linear PRCGs, the sub-class of the PRCGs which is equivalent to LCFRSs, the standard algorithm works perfectly well with input DAGs, since a valid instantiation of an argument of a predicate in a clause by some range $\langle p..q\rangle$ means that there exists (at least) one path between p and q which is recognized.

The paper will now concentrate on non-linear PRCGs, and will present a new valid parsing algorithm and study its complexities (in space and time).

In order to simplify the presentation we introduce this algorithm as a post-processing pass which will work on the shared parse-forest output by the (slightly modified) standard algorithm which accepts DAGs as input.

3.2 Parsing DAGs with non-linear PRCGs

The standard parsing algorithm of (Boullier, 2004) working on a string w can be sketched as follows. It uses a single memoized boolean function *predicate*$(A, \vec{\rho})$ where A is a predicate and $\vec{\rho}$ is a vector of ranges whose dimension is *arity*(A). The initial call to that function has the form *predicate* $(S, \langle 0..|w|\rangle)$. Its purpose is, for each A_0-clause, to instantiate each of its symbols in a consistant way. For example if we assume that the ith argument of the LHS of the current A_0-clause is $\alpha'_i X a Y \alpha''_i$ and that the ith component of $\vec{\rho_0}$ is the range $\langle p_i..q_i\rangle$ an instantiation of X, a an Y by the ranges $\langle p_X..q_X\rangle$, $\langle p_a..q_a\rangle$ and $\langle p_Y..q_Y\rangle$ is such that we have $p_i \leq p_X \leq q_X = p_a < q_a = p_a + 1 = p_Y \leq q_Y \leq q_i$ and $w = w'aw''$ with $|w'| = p_a$. Since the PRCG is non bottom-up erasing, the instantiation of all the LHS symbols implies that all the arguments of the RHS predicates A_i are also instantiated and gathered into the vector of ranges $\vec{\rho_i}$. Now, for each i ($1 \leq i \leq |RHS|$), we can call *predicate* $(A_i, \vec{\rho_i})$. If all these calls succeed, the instantiated clause can be stored as a component of the shared parse forest.[7]

In the case of a DAG $D = (Q, \Sigma, \delta, q_0, F)$ as input, there are two slight modifications, the initial call is changed by the conjunctive call *predicate*$(S, \langle q_0..f_1\rangle) \vee \ldots \vee$ *predicate* $(S, \langle q_0..f_{|F|}\rangle)$ with $f_i \in F$[8] and the terminal symbol a can be instantiated by the range $\langle p_a..q_a\rangle_D$ only if (p_a, a, q_a)

[6] In this forest oval nodes denote different instantiated predicates, while its associated instantiated clauses are presented as its daughter(s) and are denoted by square nodes. The LHS of each instantiated clause shows the instantiation of its LHS symbols. The RHS is the corresponding sequence of instantiated predicates. The number of daughters of each square node is the number of its RHS instantiated predicates.

[7] Note that such an instantiated clause could be unreachable from the (future) instantiated start symbol which will be the axiom of the shared forest considered as a CFG.

[8] Technically, each of these calls produces a forest. These individual forests may share subparts but their roots are all different. In order to have a true forest, we introduce a new root, the *super-root* whose daughters are the individual forests.

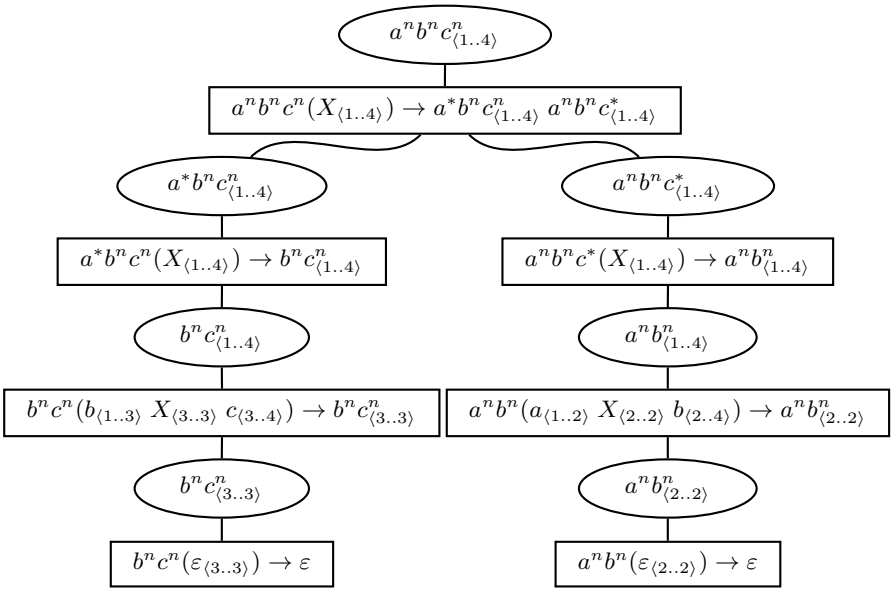

Figure 2: Parse forest for the input DAG $ab|bc$.

is a transition in δ. The variable symbol X can be instantiated by the range $\langle p_X..q_X\rangle_D$ only if $\langle p_X..q_X\rangle_D$ is valid.

3.3 Forest Filtering

We assume here that for a given PRCG G we have built the parse forest of an input DAG D as explained above and that each instantiated clause of that forest contains the range $\langle p_X..q_X\rangle_D$ of each of its instantiated symbols X. We have seen in Example 1 that this parse forest is valid if G is linear but may well be unvalid if G is non-linear. In that latter case, this happens because the range $\langle p_X..q_X\rangle_D$ of each instantiation of the non-linear variable X selects the whole sub-DAG $D_{\langle p_X..q_X\rangle}$ while each instantiation should only select a sub-language of $\mathcal{L}(D_{\langle p_X..q_X\rangle})$. For each occurrence of X in the LHS or RHS of a non-linear clause, its sub-languages could of course be different from the others. In fact, we are interested in their intersections: If their intersections are non empty, this is the language which will be associated with $\langle p_X..q_X\rangle_D$, otherwise, if their intersections are empty, then the instantiation of the considered clause fails and must thus be removed from the forest. Of course, we will consider that the language (a finite number of strings) associated with each occurrence of each instantiated symbol is represented by a DAG.

The idea of the forest filtering algorithm is to first compute the DAGs associated with each argument of each instantiated predicate during a bottom-up walk. These DAGs are called *decorations*. This processing will perform DAG compositions (including intersections, as suggested above), and will erase clauses in which empty intersections occur. If the DAG associated with the single argument of the super-root is empty, then parsing failed.

Otherwise, a top-down walk is launched (see below), which may also erase non-valid instantiated clauses. If necessary, the algorithm is completed by a classical CFG algorithm which erase non productive and unreachable symbols leaving a *reduced* grammar/forest.

In order to simplify our presentation we will assume that the PRCGs are non-combinatorial and bottom-up non-erasing. However, we can note that the following algorithm can be generalized in order to handle combinatorial PRCGs and in particular with overlapping arguments.[9] Moreover, we will assume that the forest is non cyclic (or equivalently that all cycles have previously been removed).[10]

[9]For example the non-linear combinatorial clause $A(XYZ) \rightarrow B(XY)\ B(YZ)$ has overlapping arguments.

[10]By a classical algorithm from the CFG technology.

3.3.1 The Bottom-Up Walk

For this principle algorithm, we assume that for each instantiated clause in the forest, a DAG will be associated with each occurrence of each instantiated symbol. More precisely, for a given instantiated A_0-clause, the DAGs associated with the RHS symbol occurrences are composed (see below) to build up DAGs which will be associated with each argument of its LHS predicate. For each LHS argument, this composition is directed by the sequence of symbols in the argument itself.

The forest is walked bottom-up starting from its leaves. The constraint being that an instantiated clause is visited if and only if all its RHS instantiated predicates have already all been visited (computed). This constraint can be satisfied for any non-cyclic forest.

To be more precise, consider an instantiation $c_\rho = A_0(\vec{\rho_0}) \rightarrow A_1(\vec{\rho_1}) \dots A_p(\vec{\rho_p})$ of the clause $c = A_0(\vec{\alpha_0}) \rightarrow A_1(\vec{\alpha_1}) \dots A_m(\vec{\alpha_m})$, we perform the following sequence:

1. If the clause is not top-down linear (i.e., there exist multiple occurrences of the same variables in its RHS arguments), for such variable X let the range $\langle p_X..q_X \rangle$ be its instantiation (by definition, all occurrences are instantiated by the same range), we perform the intersection of the DAGs associated with each instantiated predicate argument X. If one intersection results in an empty DAG, the instantiated clause is removed from the forest. Otherwise, we perform the following steps.

2. If a RHS variable Y is linear, it occurs once in the jth argument of predicate A_i. We perform a brand new copy of the DAG associated with the jth argument of the instantiation of A_i.

3. At that moment, all instantiated variables which occur in c_ρ are associated with a DAG. For each occurrence of a terminal symbol t in the LHS arguments we associate a (new) DAG whose only transition is (p, t, q) where p and q are brand new states with, of course, $p < q$.

4. Here, all symbols (terminals or variables) are associated with disjoints DAGs. For each LHS argument $\vec{\alpha_0}[i] = X_1^i \dots X_j^i \dots X_{p_i}^i$, we associate a new DAG which is the concatenation of the DAGs associated with the symbols $X_1^i, \dots, X_j^i, \dots$ and $X_{p_i}^i$.

5. Here each LHS argument of c_ρ is associated with a non empty DAG, we then report the individual contribution of c_ρ into the (already computed) DAGs associated with the arguments of its LHS $A_0(\vec{\rho_0})$. The DAG associated with the ith argument of $A_0(\vec{\rho_0})$ is the union (or a copy if it is the first time) of its previous DAG value with the DAG associated with the ith argument of the LHS of c_ρ.

This bottom-up walk ends on the super-root with a final decoration say R. In fact, during this bottom-up walk, we have computed the intersection of the languages defined by the input DAG and by the PRCG (i.e., we have $\mathcal{L}(R) = \mathcal{L}(D) \cap \mathcal{L}(G)$).

Example 2.

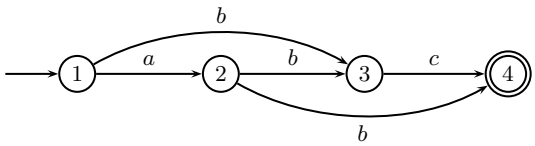

Figure 3: Input DAG associated with $abc|ab|bc$.

With the PRCG of Example 1 and the input DAG of Figure 3, we get the parse forest of Figure 4 whose transitions are decorated by the DAGs computed by the bottom-up algorithm.[11] *The crucial point to note here is the intersection which is performed between* $\{abc, bc\}$ *and* $\{abc, ab\}$ *on* $\boxed{a^n b^n c^n (X_{\langle 1..4 \rangle}) \rightarrow a^* b^n c^n_{\langle 1..4 \rangle}\ a^n b^n c^*_{\langle 1..4 \rangle}}$. *The non-empty set* $\{abc\}$ *is the final result assigned to the instantiated start symbol. Since this result is non empty, it shows that the input DAG* D *is recognized by* G*. More precisely, this shows that the sub-language of* D *which is recognized by* G *is* $\{abc\}$.

However, as shown in the previous example, the (undecorated) parse forest is not the forest built for the DAG $\mathcal{L}(D) \cap \mathcal{L}(G)$ since it may contain non-valid parts (e.g., the transitions labelled $\{bc\}$ or $\{ab\}$ in our example). In order to get the

[11] For readability reasons these DAGs are represented by their languages (i.e., set of strings). Bottom-up transitions from instantiated clauses to instantiated predicates reflects the computations performed by that instantiated clause while bottom-up transitions from instantiated predicates to instantiated clauses are the union of the DAGs entering that instantiated predicate.

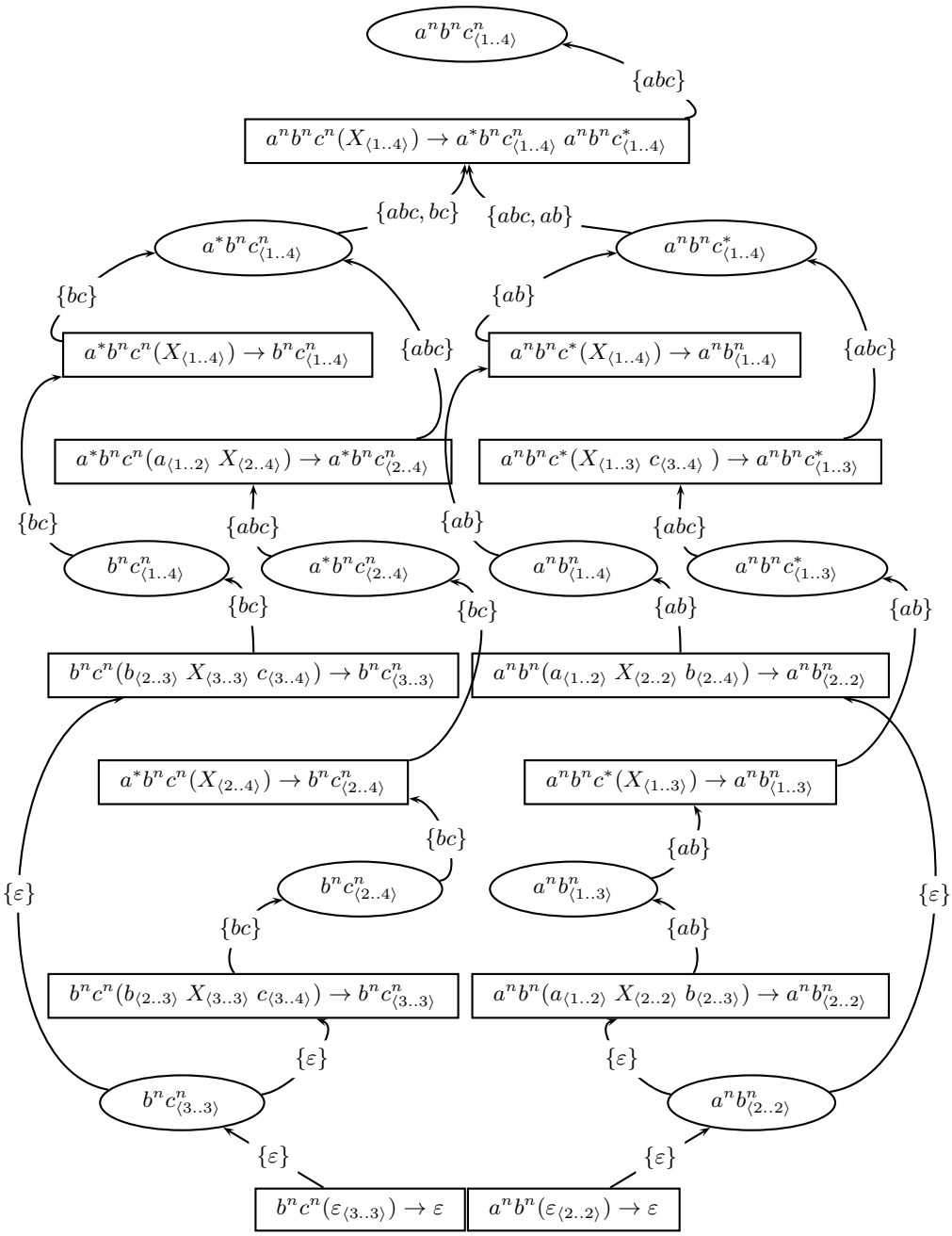

Figure 4: Bottom-up decorated parse forest for the input DAG $abc|ab|bc$.

right forest (i.e., to get a PRCG parser — not a recognizer — which accepts a DAG as input) we need to perform another walk on the previous decorated forest.

3.3.2 The Top-Down Walk

The idea of the top-down walk on the parse forest decorated by the bottom-up walk is to (re)compute all the previous decorations starting from the bottom-up decoration associated with the instantiated start predicate. It is to be noted that (the language defined by) each top-down decoration is a subset of its bottom-up counterpart. However, when a top-down decoration becomes empty, the corresponding subtree must be erased from the forest. If the bottom-up walk succeeds, we are sure that the top-down walk will not result in an empty forest. Moreover, if we perform a new bottom-up walk on this reduced forest, the new bottom-up decorations will denote the same language as their top-down decorations counterpart.

The forest is walked top-down starting from the super-root. The constraint being that an instantiated $A(\vec{\rho})$-clause is visited if and only if all the occurrences of $A(\vec{\rho})$ occurring in the RHS of instantiated clauses have all already been visited. This constraint can be satisfied for any non-cyclic forest.

Initially, we assume that each argument of each instantiated predicate has an empty decoration, except for the argument of the super-root which is decorated by the DAG R computed by the bottom-up pass.

Now, assume that a top-down decoration has been (fully) computed for each argument of the instantiated predicate $A_0(\vec{\rho_0})$. For each instantiated clause of the form $c_\rho = A_0(\vec{\rho_0}) \rightarrow A_1(\vec{\rho_1}) \ldots A_i(\vec{\rho_i}) \ldots A_m(\vec{\rho_m})$, we perform the following sequence:[12]

1. We perform the intersection of the top-down decoration of each argument of $A_0(\vec{\rho_0})$ with the decoration computed by the bottom-up pass for the same argument of the LHS predicate of c_ρ. If the result is empty, c_ρ is erased from the forest.

2. For each LHS argument, the previous results are dispatched over the symbols of this argument.[13] Thus, each instantiated LHS symbol occurrence is decorated by its own DAG. If the considered clause has several occurrences of the same variable in the LHS arguments (i.e., is bottom-up non-linear), we perform the intersection of these DAGs in order to leave a single decoration per instantiated variable. If an intersection results in an empty DAG, the current clause is erased from the forest.

3. The LHS instantiated variable decorations are propagated to the RHS arguments. This propagation may result in DAG concatenations when a RHS argument is made up of several variables (i.e., is combinatorial).

4. At last, we associate to each argument of $A_i(\vec{\rho_i})$ a new decoration which is computed as the union of its previous top-down decoration with the decoration just computed.

Example 3. *When we apply the previous algorithm to the bottom-up parse forest of Example 2, we get the top-down parse forest of Figure 5. In this parse forest, erased parts are laid out in light gray. The more noticable points w.r.t. the bottom-up forest are the decorations between* $\boxed{a^n b^n c^n (X_{\langle 1..4 \rangle}) \rightarrow a^* b^n c^n_{\langle 1..4 \rangle}\ a^n b^n c^*_{\langle 1..4 \rangle}}$ *and its RHS predicates* $a^* b^n c^n_{\langle 1..4 \rangle}$ *and* $a^n b^n c^*_{\langle 1..4 \rangle}$ *which are changed both to* $\{abc\}$ *instead of* $\{abc, bc\}$ *and* $\{abc, ab\}$. *These two changes induce the indicated erasings.*

[12] The decoration of each argument of $A_i(\vec{\rho_i})$ is either initially empty or has already been partially computed.

[13] Assume that $\vec{\rho_0}[k] = \langle p..q \rangle_D$, that the decoration DAG associated with the kth argument of $A_0(\vec{\rho_0})$ is $D'_{\langle p..q \rangle} = (Q'_{\langle p..q \rangle}, \Sigma_{\langle p..q \rangle}, \delta'_{\langle p..q \rangle}, p', F'_{\langle p..q \rangle})$ (we have $\mathcal{L}(D'_{\langle p..q \rangle}) \subseteq \mathcal{L}(D_{\langle p..q \rangle})$) and that $\vec{\alpha_0}[k] = \alpha_k^1 X \alpha_k^2$ and that $\langle i..j \rangle_D$ is the instantiation of the symbol X in c_ρ. Our goal is to extract from $D'_{\langle p..q \rangle}$ the decoration DAG $D'_{\langle i..j \rangle}$ associated with that instantiated occurrence of X. This computation can be helped if we maintain, associated with each decoration DAG a function, say d, which maps each state of the decoration DAG to a set of states (bounds) of the input DAG D. If, as we have assumed, D is minimal, each set of states is a singleton, we can write $d(p') = p$, $d(f') = q$ for all $f' \in F'_{\langle p..q \rangle}$ and more generally $d(i') \in Q$ if $i' \in Q'$. Let $I' = \{i' \mid i' \in Q'_{\langle p..q \rangle} \wedge d(i') = i\}$ and $J' = \{j' \mid j' \in Q'_{\langle p..q \rangle} \wedge d(j') = j\}$. The decoration DAG $D'_{\langle i..j \rangle}$ is such that $\mathcal{L}(D'_{\langle i..j \rangle}) = \bigcup_{i' \in I', j' \in J'} \{x \mid x \text{ is a path from } i' \text{ to } j'\}$. Of course, together with the construction of $D'_{\langle i..j \rangle}$, its associated function d must also be built.

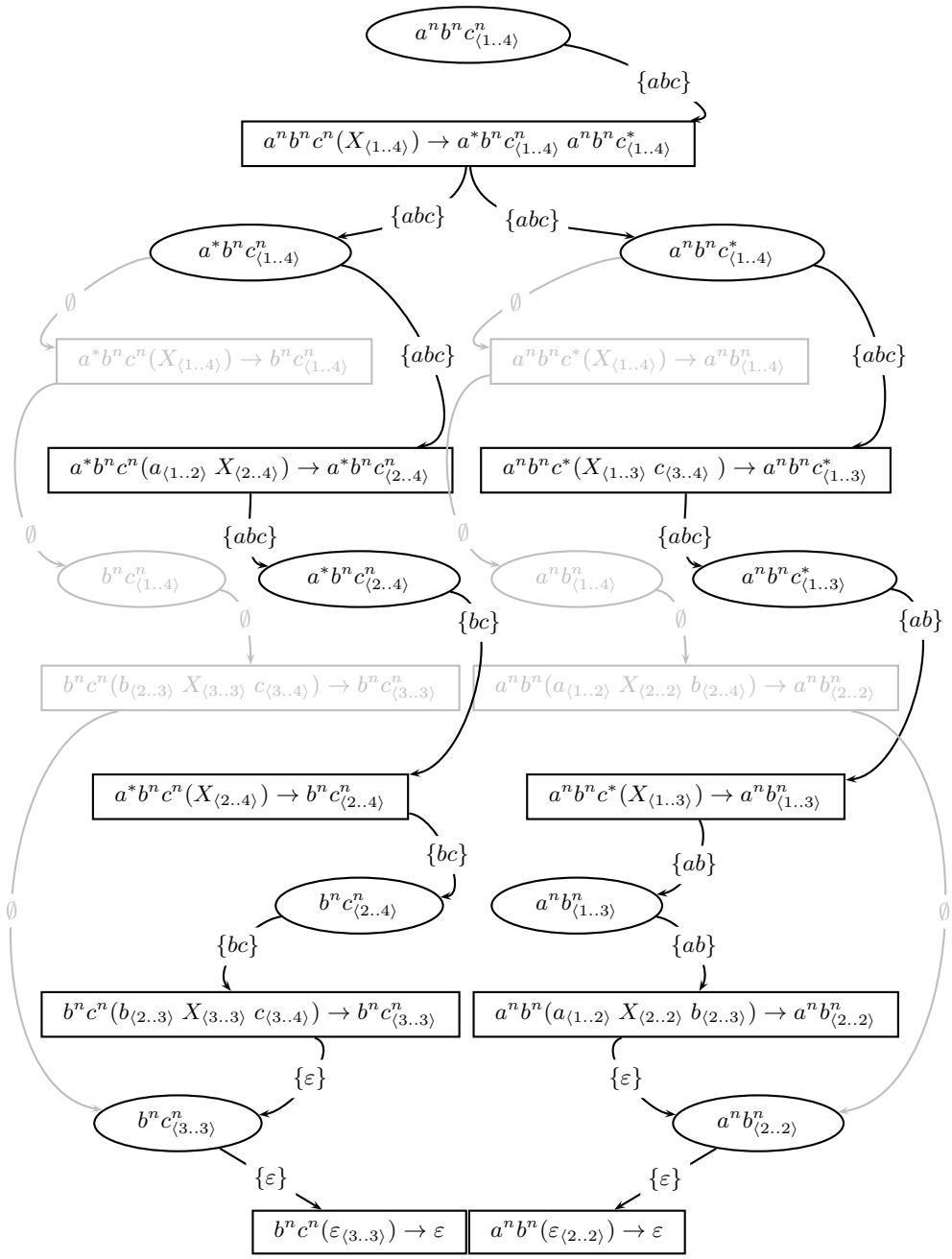

Figure 5: Top-down decorated parse forest for the input DAG $abc|ab|bc$.

3.4 Time and Space Complexities

In this Section we study the time and size complexities of the forest filtering algorithm.

Let us consider the sub-DAG $D_{\langle p..q \rangle}$ of the minimal input DAG D and consider any (finite) regular language $L \subseteq \mathcal{L}(D_{\langle p..q \rangle})$, and let D_L be the minimal DAG s.t. $\mathcal{L}(D_L) = L$. We show, on an example, that $|D_L|$ may be an exponential w.r.t. $|D_{\langle p..q \rangle}|$.

Consider, for a given $h > 0$, the language $(a|b)^h$. We know that this language can be represented by the minimal DAG with $h+1$ states of Figure 6.

Assume that $h = 2k$ and consider the sub-language L_{2k} of $(a|b)^{2k}$ (nested well-parenthesized strings) which is defined by

1. $L_2 = \{aa, bb\}$;
2. $k > 1$, $L_{2k} = \{axa, bxb \mid x \in L_{2k-2}\}$,

It is not difficult to see that the DAG in Figure 7 defines L_{2k} and is minimal, but its size $2^{k+2} - 2$ is an exponential in the size $2k+1$ of the minimal DAG for the language $(a|b)^{2k}$.

This results shows that, there exist cases in which some minimal DAGs D' that define sub-languages of minimal DAGs D may have a exponential size (i.e., $|D'| = \mathcal{O}(2^{|D|})$. In other words, when, during the bottom-up or top-down walk, we compute union of DAGs, we may fall on these pathologic DAGs that will induce a combinatorial explosion in both time and space.

3.5 Implementation Issues

Of course, many improvements may be brought to the previous principle algorithms in practical implementations. Let us cite two of them. First it is possible to restrict the number of DAG copies: a DAG copy is not useful if it is the last reference to that DAG.

We shall here develop the second point on a little more: if an argument of a predicate is never *used* in an clause involving non-linearity, it is only a waste of time to compute its decoration. We say that A^k, the kth argument of the predicate A is a *non-linear predicate argument* if there exists a clause c in which A occurs in the RHS and whose k^{th} argument has at least one common variable another argument B^h of some predicate B of the RHS (if $B = A$, then of course k and h must be different). It is clear that B^h is then non-linear as well. It is not difficult to see that decorations needs only to be computed if they are associated with a non-linear predicate argument. It is possible to compute those non-linear predicate arguments statically (when building the parser) when the PRCG is defined within a single module. However, if the PRCG is given in several modules, this full static computation is no longer possible. The non-linear predicate arguments must thus be identified at parse time, when the whole grammar is available. This rather trivial algorithm will not be described here, but it should be noted that it is worth doing since in practice it prevents decoration computations which can take an exponential time.

4 Conclusion

In this paper we have shown how PRCGs can handle DAGs as an input. If we consider the linear PRCG, the one equivalent to LCFRS, the parsing time remains polynomial. Moreover, input DAGs necessitate only rather cosmetic modifications in the standard parser.

In the non-linear case, the standard parser may produce illegal parses in its output shared parse forest. It may even produce a (non-empty) shared parse forest though no sentences of the input DAG are in the language defined by our non-linear PRCG. We have proposed a method which uses the (slightly modified) standard parser but prunes, within extra passes, its output forest and leaves all and only valid parses. During these extra bottom-up and top-down walks, this pruning involves the computation of finite languages by means of concatenation, union and intersection operations. The sentences of these finite languages are always substrings of the words of the input DAG D. We choose to represent these intermediate finite languages by DAGs instead of sets of strings because the size of a DAG is, at worst, of the same order as the size of a set of strings but it could, in some cases, be exponentially smaller.

However, the time taken by this extra pruning pass cannot be guaranteed to be polynomial, as expected from previously known complexity results (Bertsch and Nederhof, 2001). We have shown an example in which pruning takes an exponential time and space in the size of D. The deep reason comes from the fact that if L is a finite (regular) language defined by some minimal DAG D, there are cases where a sub-language of

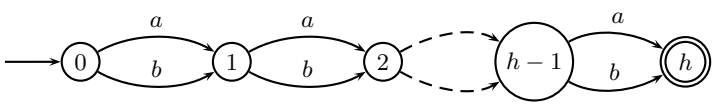

Figure 6: Input DAG associated with the language $(a|b)^h$, $h > 0$.

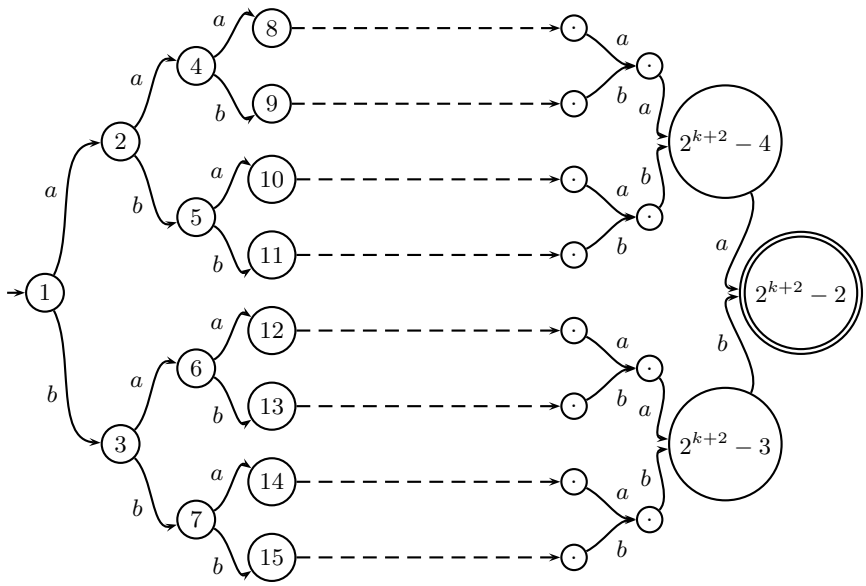

Figure 7: DAG associated with the language of nested well-parenthesized strings of length $2k$.

L may require to be defined by a DAG whose size is an exponential in the size of D. Of course this combinatorial explosion is not a fatality, and we may wonder whether, in the particular case of NLP it will practically occur?

References

Franois Barthélemy, Pierre Boullier, Philippe Deschamp, and Éric de la Clergerie. 2001. Guided parsing of range concatenation languages. In *Proceedings of the 39th Annual Meeting of the Association for Comput. Linguist. (ACL'01)*, pages 42–49, University of Toulouse, France.

Eberhard Bertsch and Mark-Jan Nederhof. 2001. On the complexity of some extensions of rcg parsing. In *Proceedings of IWPT'01*, Beijing, China.

Pierre Boullier, 2004. *New Developments in Parsing Technology*, volume 23 of *Text, Speech and Language Technology*, chapter Range Concatenation Grammars, pages 269–289. Kluwer Academic Publishers, H. Bunt, J. Carroll, and G. Satta edition.

Jeffrey D. Hopcroft and John E. Ullman. 1979. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Mass.

Bernard Lang. 1994. Recognition can be harder than parsing. *Computational Intelligence*, 10(4):486–494.

K. Vijay-Shanker, David Weir, and Aravind K. Joshi. 1987. Characterizing structural descriptions produced by various grammatical formalisms. In *Proceedings of the 25th Meeting of the Association for Comput. Linguist. (ACL'87)*, pages 104–111, Stanford University, CA.