Generalized Predictive Shift-Reduce Parsing for Hyperedge Replacement Graph Grammars

Berthold Hoffmann[1] and Mark Minas[2](✉)

[1] Universität Bremen, Bremen, Germany
hof@informatik.uni-bremen.de
[2] Universität der Bundeswehr München, Neubiberg, Germany
mark.minas@unibw.de

Abstract. Parsing for graph grammars based on hyperedge replacement (HR) is in general NP-hard, even for a particular grammar. The recently developed predictive shift-reduce (PSR) parsing is efficient, but restricted to a subclass of unambiguous HR grammars. We have implemented a generalized PSR parsing algorithm that applies to all HR grammars, and pursues severals parses in parallel whenever decision conflicts occur. We compare GPSR parsers with the Cocke-Younger-Kasami parser and show that a GPSR parser, despite its exponential worst-case complexity, can be much faster.

Keywords: Hyperedge replacement grammar · Graph parsing

1 Introduction

It is well known that parsing for graph grammars based on hyperedge replacement (HR) is in general NP-hard, even for a particular grammar [8]. In earlier work [6], we have devised predictive shift-reduce parsing (PSR), which lifts Knuth's LR string parsing [9] to graphs, is efficient, but unfortunately restricted to a subclass of unambiguous HR grammars. This makes it unsuitable for applications in natural language processing (NLP) where grammars are often ambiguous. So we extend the PSR algorithm to arbitrary HR grammars in this paper: Just like Tomita's generalized LR string parser [12], the *generalized PSR parser* pursues all possible parses of a graph in parallel whenever ambiguity occurs. We describe the implementation of the generalized PSR parser by Mark Minas,[1] and compare its efficiency with the Cocke-Younger-Kasami parser for arbitrary HR grammars [11].

The remainder of this paper is structured as follows. After recalling HR grammars in Sect. 2 and PSR parsing in Sect. 3, we introduce generalized PSR parsing in Sect. 4, and compare its performance with CYK parsing in Sect. 5.

[1] In the graph parser generator *Grappa*, available at www.unibw.de/inf2/grappa.

© Springer Nature Switzerland AG 2019
C. Martín-Vide et al. (Eds.): LATA 2019, LNCS 11417, pp. 233–245, 2019.
https://doi.org/10.1007/978-3-030-13435-8_17

Due to lack of space, our presentation is driven by a small example—a grammar for series-parallel graphs—that exhibits many peculiarities of generalized PSR parsing. In Sect. 6, we conclude by indicating related and future work.

2 Graph Grammars Based on Hyperedge Replacement

Throughout the paper, we assume that X is a global, countably infinite supply of *nodes*, and that Σ is a finite set of *symbols* that comes with an *arity function* $arity\colon \Sigma \to \mathbb{N}$, and is partitioned into disjoint subsets $\mathcal{N}$ of *nonterminals* and $\mathcal{T}$ of *terminals*.

We represent hypergraphs as ordered sequences of edge literals, where each literal represents an edge with its attached nodes. This is convenient as we shall derive (and parse) the edges of a graph in a fixed order.

Definition 1 (Hypergraph). For a symbol $\mathsf{a} \in \Sigma$ and $k = arity(\mathsf{a})$ pairwise distinct nodes $x_1, \ldots, x_k \in X$, $a = \mathsf{a}(x_1, \ldots, x_k)$ represents a *hyperedge* that is labeled with a and attached to $x_1, \ldots, x_k$. $\mathcal{E}_\Sigma$ denotes the set of hyperedges (over Σ).

A *hypergraph* $\langle \gamma, V \rangle$ consists of a sequence $\gamma = e_1 \cdots e_n \in \mathcal{E}_\Sigma^*$ of *hyperedges* and a finite set $V \subseteq X$ of *nodes* that contains all nodes attached to the hyperedges of γ. $\mathcal{G}_\Sigma$ denotes the set of all hypergraphs (over Σ).

In the following, we usually call hypergraphs just *graphs* and hyperedges just *edges*. Moreover, we denote a graph just by its edges γ, and refer to its nodes by V_γ.[2] The "concatenation" of two graphs $\alpha, \beta \in \mathcal{G}_\Sigma$ yields a graph $\gamma = \alpha\beta$ with nodes $V_\gamma = V_\alpha \cup V_\beta$. Two graphs γ and γ' are *equivalent*, written $\gamma \bowtie \gamma'$, if $V_\gamma = V_{\gamma'}$ and γ is a permutation of γ'.

Note that we order the edges of a graph in rules and derivations. However, the relation $\bowtie$ makes graphs with permuted edges equivalent, like in ordinary definitions of graphs. Our parsers will make sure that equivalent graphs are always processed in the same way.

An injective function $\varrho\colon X \to X$ is called a *renaming*, and γ^ϱ denotes the graph obtained by replacing all nodes in γ (and in V_γ) according to ϱ. A *hyperedge replacement rule* $r = (A \to \alpha)$ (*rule* for short) has a nonterminal edge $A \in \mathcal{E}_\mathcal{N}$ as its *left-hand side*, and a graph $\alpha \in \mathcal{G}_\Sigma$ with $V_A \subseteq V_\alpha$ as its *right-hand side*.

Consider a graph $\gamma = \beta\bar{A}\bar{\beta} \in \mathcal{G}_\Sigma$ with a nonterminal edge $\bar{A}$ and a rule $r = (A \to \alpha)$. A renaming $\mu\colon X \to X$ is a *match* (of r in γ) if $A^\mu = \bar{A}$ and if $V_\gamma \cap V_{\alpha^\mu} \subseteq V_{A^\mu}$.[3] A match μ of r *derives* γ to the graph $\gamma' = \beta\alpha^\mu\bar{\beta}$. This is denoted as $\gamma \Rightarrow_{r,\mu} \gamma'$. If $\mathcal{R}$ is a finite set of rules, we write $\gamma \Rightarrow_\mathcal{R} \gamma'$ if $\gamma \Rightarrow_{r,\mu} \gamma'$ for some match μ of some rule $r \in \mathcal{R}$.

Definition 2 (HR Grammar). A *hyperedge replacement grammar* $\Gamma = (\Sigma, \mathcal{T}, \mathcal{R}, Z)$ (*HR grammar* for short) consists of *symbols* Σ with *terminals*

[2] V_γ may contain *isolated* nodes that are not attached to any edge in γ.

[3] I.e., a match μ makes sure that the nodes of α^μ that do not occur in $\bar{A} = A^\mu$ do not collide with the other nodes in γ.

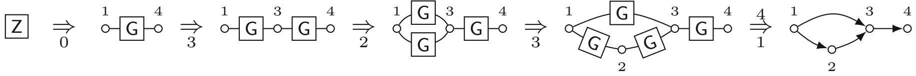


Fig. 1. *A derivation of the graph* $\mathsf{e}(1,3)\,\mathsf{e}(1,2)\,\mathsf{e}(2,3)\,\mathsf{e}(3,4)$. Nodes are drawn as circles, nonterminal edges as boxes around their label, with lines to their attached nodes, and terminal edges as arrows from their first to their second attached node; since e is the only terminal label, we omitted it in the terminal graph.

$\mathcal{T} \subseteq \Sigma$ as assumed above, a finite set $\mathcal{R}$ of rules, and a *start graph* $Z = \mathsf{Z}()$ with $\mathsf{Z} \in \mathcal{N}$ of arity 0. Γ generates the language

$$\mathcal{L}(\Gamma) = \{g' \in \mathcal{G}_{\mathcal{T}} \mid Z \Rightarrow^*_{\mathcal{R}} g, g' \bowtie g\}.$$

In the following, we simply write $\Rightarrow$ and $\Rightarrow^*$ because the rule set $\mathcal{R}$ in question will always be clear from the context.

Example 1 (A HR Grammar for Series-Parallel Graphs). The following rules

$$\mathsf{Z}() \underset{0}{\to} \mathsf{G}(x,y) \qquad \mathsf{G}(x,y) \underset{1}{\to} \mathsf{e}(x,y)$$
$$\mathsf{G}(x,y) \underset{2}{\to} \mathsf{G}(x,y)\,\mathsf{G}(x,y) \qquad \mathsf{G}(x,y) \underset{3}{\to} \mathsf{G}(x,z)\,\mathsf{G}(z,y)$$

generate series-parallel graphs [8, p. 99]; see Fig. 1 for a derivation with graphs drawn as diagrams.

3 Predictive Shift-Reduce Parsing

The article [6] gives detailed definitions and correctness proofs for PSR parsing. Here we recall the concepts only so far that we can describe its generalization in the next section.

A PSR parser attempts to construct a derivation by reading the edges of a given input graph one after the other.[4] However, the parser must not assume that the edges of the input graph come in the same order as in a derivation. E.g., when constructing the derivation in Fig. 1, it must also accept an input graph $\mathsf{e}(2,3)\,\mathsf{e}(1,2)\,\mathsf{e}(1,3)\,\mathsf{e}(3,4)$ where the edges are permuted.

Before parsing starts, a procedure described in [5, Sect. 4] analyzes the grammar for the *unique start node* property, by computing the possible incidences of all nodes created by a grammar. The unique start nodes have to be matched by some nodes in the start rule of the grammar, thus determining where parsing begins. For our example, the procedure detects that every series-parallel graph has a unique root (without ingoing edges), and that the node x in the start rule

[4] We silently assume that input graphs do not have isolated nodes. This is no real restriction as one can add special edges to such nodes.

$\mathsf{Z}() \to \mathsf{G}(x, y)$ must be bound to the root of any input graph.[5] If the input graph has no root, or more than one, it cannot be series-parallel, so that parsing fails immediately.

A PSR parser is a push-down automaton that is controlled by a *characteristic finite automaton* (CFA). The stack of the PSR parser consists of states of the CFA. The parser makes sure that the sequence of states on its stack always describes a valid walk through its CFA. In order to do so, the parser generator computes a *parsing table* with processing instructions that control the parser.

Table 1 shows the parsing table for our example of series-parallel graphs. It has been generated by the graph parser generator *Grappa* (see footnote 1), using the constructions described in [6]. The rows of the table correspond to states of the CFA. Each of these states has a certain number of parameters. For instance, $Q_2(p, q)$ has two parameters p and q. Parameters remain *abstract* in the CFA and in the parsing table; only the parser will bind them to nodes of the input graph, and store them in *concrete* states on its stack.

When the parser starts, nothing of its input graph has been read yet, and its stack consists of a single concrete state Q_0, where its parameter p is bound to the unique start node, namely the root of the input graph.

The columns of the table correspond to terminal and nonterminal edges as well as the end-of-input marker \$. Column $\mathsf{e}(x, y)$ in Table 1 contains all actions that can be taken by the parser if the input graph contains an edge $\mathsf{e}(x, y)$ that is still unread. Column \$ contains the actions to be done if the input graph has been read completely. We will come to column $\mathsf{G}(x, y)$ later.

The parser looks up its next action in the parsing table by inspecting the top-most state on its stack and all unread edges of the input graph. For illustration, let us assume that the parser has state Q_3 on top of its stack, with its parameters p and q being bound (by an appropriate renaming σ) to the input graph nodes p^σ and q^σ, resp., so that the parser must look into row $Q_3(p, q)$. If the input graph contains an unread e-labeled edge, the entry in column $\mathsf{e}(x, y)$ applies. The corresponding table entry contains two possible actions, a shift and a reduce.

The *shift* operation can be selected if the input graph contains an unread e-labeled edge e that connects p^σ, either with q^σ, or with any node that has not yet occurred in the parse, indicated by "$-$" in the condition. If this shift operation is selected, it marks e as read and pushes a new concrete state Q_1 onto the stack, where the parameters p and q of Q_1 are bound to the source and the target node of e.

The *reduce* operation does not require any further condition to be satisfied. It consists of two steps: First, it pops as many states from the top of the stack as the right-hand side of the rule has edges. In our example "reduce 2, $\mathsf{G}(p, q)$" refers to rule 2, with two edges in its right-hand side. Second, the reduce operation looks up the row for the new top-most state of the stack, selects the operation for

[5] Actually, series-parallel graphs do also have a unique sink (without outgoing edges), which could be used as a second start node bound to y. However, this variation of the grammar would exhibit less peculiarities of the GPSR parser.

Table 1. PSR parsing table for series-parallel graphs.

State	$\mathsf{e}(x, y)$	$	$\mathsf{G}(x, y)$
$Q_0(p)$	shift $Q_1(x, y)$ if $(x, y) = (p, -)$	error	goto $Q_2(x, y)$ if $(x, y) = (p, -)$
$Q_1(p, q)$	reduce 1, $\mathsf{G}(p, q)$		error
$Q_2(p, q)$	shift $Q_1(x, y)$ if $(x, y) = (p, q)$ or $(x, y) = (p, -)$ or $(x, y) = (q, -)$	accept	goto $Q_3(x, y)$ if $(x, y) = (p, q)$ goto $Q_4(x, y, q)$ if $(x, y) = (p, -)$ goto $Q_5(x, y, p)$ if $(x, y) = (q, -)$
$Q_3(p, q)$	shift $Q_1(x, y)$ if $(x, y) = (p, q)$ or $(x, y) = (p, -)$ reduce 2, $\mathsf{G}(p, q)$	reduce 2, G(p,q)	goto $Q_3(x, y)$ if $(x, y) = (p, q)$ goto $Q_4(x, y, q)$ if $(x, y) = (p, -)$
$Q_4(p, q, u)$	shift $Q_1(x, y)$ if $(x, y) = (p, q)$ or $(x, y) = (q, u)$ or $(x, y) = (p, -)$ or $(x, y) = (q, -)$	error	goto $Q_3(x, y)$ if $(x, y) = (p, q)$ goto $Q_4(x, y, q)$ if $(x, y) = (p, -)$ goto $Q_6(x, y, p)$ if $(x, y) = (q, u)$ goto $Q_7(x, y, u, p)$ if $(x, y) = (q, -)$
$Q_5(p, q, u)$	shift $Q_1(x, y)$ if $(x, y) = (p, q)$ or $(x, y) = (p, -)$ or $(x, y) = (q, -)$ reduce 3, $\mathsf{G}(u, q)$	reduce 3, $\mathsf{G}(u, q)$	goto $Q_3(x, y)$ if $(x, y) = (p, q)$ goto $Q_4(x, y, q)$ if $(x, y) = (p, -)$ goto $Q_5(x, y, p)$ if $(x, y) = (q, -)$
$Q_6(p, q, u)$	shift $Q_1(x, y)$ if $(x, y) = (p, q)$ or $(x, y) = (p, -)$ reduce 3, $\mathsf{G}(u, q)$	reduce 3, $\mathsf{G}(u, q)$	goto $Q_3(x, y)$ if $(x, y) = (p, q)$ goto $Q_4(x, y, q)$ if $(x, y) = (p, -)$
$Q_7(p, q, u, v)$	shift $Q_1(x, y)$ if $(x, y) = (p, q)$ or $(x, y) = (q, u)$ or $(x, y) = (p, -)$ or $(x, y) = (q, -)$ reduce 3, $\mathsf{G}(v, q)$	reduce 3, $\mathsf{G}(v, q)$	goto $Q_3(x, y)$ if $(x, y) = (p, q)$ goto $Q_4(x, y, q)$ if $(x, y) = (p, -)$ goto $Q_6(x, y, p)$ if $(x, y) = (q, u)$ goto $Q_7(x, y, u, p)$ if $(x, y) = (q, -)$

the new nonterminal edge with label G that connects p^σ with q^σ, i.e., in column $\mathsf{G}(x, y)$, and pushes the corresponding state onto the stack.

The entries *accept* and *error* in column $ express that, if all edges of the input graph have been read, the parser terminates with success if the top-most state is Q_2, or with failure if it is Q_0 or Q_4.

A PSR parser must always be able to select the correct operation; it must not happen that the parser must choose between two or more operations where one of them leads to a successful parse whereas another one leads to failure. Such a situation is called a *conflict*. It is clear that a PSR parser always selects the correct action if conflicts cannot occur. However, a PSR parser does not know a priori which unread edge must be selected next. Hence, there are not only the shift-reduce or reduce-reduce conflicts (well known from LR parsing [9]). Shift-shift conflicts may also occur if the parser has to choose which input edge should be read next. Moreover, a shift alone may raise a conflict, since the unread input graph may contain more than one edge matching a pattern like $\mathsf{e}(p, -)$. Only if the *free edge choice* property holds, the parser knows that any of these edges may be processed, without affecting the result of the parse.[6]

For our example of series-parallel graphs, conflicts arise in all states, except for Q_0 and Q_1. For $Q_3(p^\sigma, q^\sigma)$, e.g., the parser can always select the reduction, but it can also shift any edge connecting p^σ with q^σ or with any other unread node. Apparently, not every choice will lead to a successful parse, even if the input graph is valid.

Thus the parsing table does not always allow to predict the next correct action, and the grammar does not have a PSR parser. This problem can be solved by generalizing PSR parsing as described in the next section.

4 Generalized Predictive Shift-Reduce Parsing

Before we describe GPSR parsing for HR grammars, let us briefly recapitulate LR(k) parsing for context-free string grammars ([9], with $k = 1$ symbols of lookahead) and how this is extended to generalized LR (GLR) parsing [12]. An LR(1) parser is controlled by a parsing table derived from the CFA of the grammar. The parsing table assigns a unique parser action to each state of the CFA and to each terminal symbol: *shift*, *reduce*, *accept*, or *error*. In each step, the parser executes the action specified for the current state on top of the stack and the next unread input symbol (the *look-ahead*). However, LR(1) parsing is not possible if the parsing table has conflicts, i.e., if there is a state q and a look-ahead symbol a associated with two actions or more. A parser that reaches q with a look-ahead symbol a has as many choices how it may continue, i.e., the parse stack can be modified in different ways. A search process must then explore which of the resulting parse stacks can be further extended to a successful parse.

A GLR parser organizes this search process as a breadth-first search. It reads the input string from left to right. At any time, it has read a certain prefix α of the input string. It maintains the set of all (parse) stacks which can be obtained by reading α. This set of stacks is in fact processed in rounds as follows: For each stack, the parser determines all possible actions based on the parsing table, the top-most state of the stack, and the look-ahead symbol. The parser has found a successful parse if the action is *accept* and the entire input string has been read.

[6] This property can be determined by the parser generator as well. However, it does not hold for the grammar of series-parallel graphs.

(It may proceed if further parses shall be found.) If the action is an *error*, the parser just discards this stack, stops if this has been the last remaining stack, and fails altogether if it has not found a successful parse previously. If the parsing table, however, indicates more than one possible action, the parser duplicates the stack for each of them, and performs each action on one of the copies. If the action is a *shift*, the resulting stack is no longer considered in this round, but only in the next one. This way, at the beginning of the next round, each stack is the result of reading the look-ahead symbol in a *shift* action, and having read the same prefix of the input string.

In fact, a GLR parser does not store complete copies of stacks, but shares their common prefixes and suffixes. The resulting structure is known as a *graph-structured stack* (GSS). An individual stack is represented as a path in the GSS, from some top-most state to the unique initial state.

A GPSR parser generalizes a PSR parser in the same way as a GLR parser generalizes an LR parser. It also maintains a set of parse stacks, which contain concrete states whose parameters are bound to nodes of the input graph. However, a GPSR parser must also deal with the fact that there is no a priori reading sequence of edges of the input graph.

This affects a GPSR parser even more than a PSR parser since a GPSR parser may be forced to pursue different reading sequences in parallel while it performs the search process. This has consequences as follows:

- Each parse stack corresponds to a specific part of the input graph that has been read already. Hence, the parser must store, for each stack separately, which edges of the input graph have been read.
  Sets of stacks are stored as a GSS like in GLR parsers. Each GSS node corresponds to a concrete state. Additionally, each GSS node keeps track of the set of input graph edges that have been read so far. Note that GSS nodes may be shared only if both their concrete states and their sets of read edges coincide.
- GPSR parsers cannot process their sets of stacks in rounds. When a stack is obtained by executing a *shift* action, the parser must not wait until the same edge has been read in all the other stacks; they may read other edges first. As a consequence, a GPSR parser needs other strategies to control the order in which stacks are processed. Strategies are discussed in Sect. 5.

We demonstrate GPSR parsing using the example of series-parallel graphs and the input graph $\mathsf{e}(1,2)\,\mathsf{e}(2,3)\,\mathsf{e}(1,3)\,\mathsf{e}(3,4)$ derived in Fig. 1. We refer to these edges by the letters a, b, c, and d. We write GSS nodes in compact form: e.g., 5^{341}_{cd} refers to the concrete state $Q_5(3,4,1)$ and indicates that the edges $c = \mathsf{e}(2,3)$ and $d = \mathsf{e}(3,4)$ have been read already.

The parser determines node 1 as the unique start node, i.e., it starts with concrete state $Q_0(1)$, with all edges of the input graph unread. So the GSS in step 0 consists of $0^1_\varnothing$ (cf. Table 2). The parsing table in Table 1 indicates that the parser can shift both edge $a = \mathsf{e}(1,2)$ and edge $c = \mathsf{e}(1,3)$, resulting in the stacks $0^1_\varnothing\ 1^{12}_a$ and $0^1_\varnothing\ 1^{13}_c$. Table 2 shows the corresponding GSS in step 1. (Note that "new" GSS nodes are set in boldface.) Step 1 continues with processing stack

Table 2. Graph-structured stacks and steps of the GPSR parser when parsing the graph consisting of the hyperedges $a = \mathsf{e}(1,2), b = \mathsf{e}(2,3), c = \mathsf{e}(1,3), d = \mathsf{e}(3,4)$.

Step	Graph-structured stack	Action
0	$\mathbf{0^1_\varnothing}$	shift a, c
1	$0^1_\varnothing \leftarrow \mathbf{1^{12}_a}$; $0^1_\varnothing \leftarrow \mathbf{1^{13}_c}$	reduce 1
2	$0^1_\varnothing \leftarrow \mathbf{2^{12}_a}$; $0^1_\varnothing \leftarrow 1^{13}_c$	reduce 1
3	$0^1_\varnothing \leftarrow 2^{12}_a$; $0^1_\varnothing \leftarrow \mathbf{2^{13}_c}$	shift b, c
4	$0^1_\varnothing \leftarrow 2^{12}_a \leftarrow \mathbf{1^{13}_{ac}}$, $\mathbf{1^{23}_{ab}}$; $0^1_\varnothing \leftarrow 2^{13}_c$	shift a, d
5	$0^1_\varnothing \leftarrow 2^{12}_a \leftarrow 1^{13}_{ac}$, 1^{23}_{ab}; $0^1_\varnothing \leftarrow 2^{13}_c \leftarrow \mathbf{1^{12}_{ac}}$, $\mathbf{1^{34}_{cd}}$	reduce 1
6	$0^1_\varnothing \leftarrow 2^{12}_a \leftarrow 1^{13}_{ac}$, $\mathbf{5^{231}_{ab}}$; $0^1_\varnothing \leftarrow 2^{13}_c \leftarrow 1^{12}_{ac}$, 1^{34}_{cd}	reduce 1
7	$0^1_\varnothing \leftarrow 2^{12}_a \leftarrow 1^{13}_{ac}$, 5^{231}_{ab}; $0^1_\varnothing \leftarrow 2^{13}_c \leftarrow 1^{12}_{ac}$, $\mathbf{5^{341}_{cd}}$	shift d reduce 3
8	$0^1_\varnothing \leftarrow 2^{12}_a \leftarrow 1^{13}_{ac}$, $5^{231}_{ab} \leftarrow \mathbf{1^{34}_{abd}}$; $0^1_\varnothing \leftarrow \mathbf{2^{13}_{ab}}$; $0^1_\varnothing \leftarrow 2^{13}_c \leftarrow 1^{12}_{ac}$, 5^{341}_{cd}	reduce 3*
9	$0^1_\varnothing \leftarrow 2^{12}_a \leftarrow 1^{13}_{ac}$, $5^{231}_{ab} \leftarrow 1^{34}_{abd}$; $0^1_\varnothing \leftarrow 2^{13}_{ab}$; $0^1_\varnothing \leftarrow 2^{13}_c \leftarrow 1^{12}_{ac}$	shift c, d
10	$0^1_\varnothing \leftarrow 2^{12}_a \leftarrow 1^{13}_{ac}$, $5^{231}_{ab} \leftarrow 1^{34}_{abd}$; $0^1_\varnothing \leftarrow 2^{13}_{ab} \leftarrow 1^{34}_{abd}$, $\mathbf{1^{13}_{abc}}$; $0^1_\varnothing \leftarrow 2^{13}_c \leftarrow 1^{12}_{ac}$	reduce 1
11	$0^1_\varnothing \leftarrow 2^{12}_a \leftarrow 1^{13}_{ac}$, $5^{231}_{ab} \leftarrow \mathbf{5^{342}_{abd}}$; $0^1_\varnothing \leftarrow 2^{13}_{ab} \leftarrow \mathbf{5^{341}_{abd}}$, 1^{13}_{abc}; $0^1_\varnothing \leftarrow 2^{13}_c \leftarrow 1^{12}_{ac}$	reduce 1
12	$0^1_\varnothing \leftarrow 2^{12}_a \leftarrow 1^{13}_{ac}$, $5^{231}_{ab} \leftarrow 5^{342}_{abd}$; $0^1_\varnothing \leftarrow 2^{13}_{ab} \leftarrow 5^{341}_{abd}$, $\mathbf{3^{13}_{abc}}$; $0^1_\varnothing \leftarrow 2^{13}_c \leftarrow 1^{12}_{ac}$	reduce 3*
13	$0^1_\varnothing \leftarrow 2^{12}_a \leftarrow 1^{13}_{ac}$; $0^1_\varnothing \leftarrow 2^{13}_{ab} \leftarrow 5^{341}_{abd}$, 3^{13}_{abc}; $0^1_\varnothing \leftarrow 2^{13}_c \leftarrow 1^{12}_{ac}$	reduce 3*
14	$0^1_\varnothing \leftarrow 2^{12}_a \leftarrow 1^{13}_{ac}$; $0^1_\varnothing \leftarrow 2^{13}_{ab} \leftarrow 3^{13}_{abc}$; $0^1_\varnothing \leftarrow 2^{13}_c \leftarrow 1^{12}_{ac}$	reduce 2
15	$0^1_\varnothing \leftarrow 2^{12}_a \leftarrow 1^{13}_{ac}$; $0^1_\varnothing \leftarrow \mathbf{2^{13}_{abc}}$; $0^1_\varnothing \leftarrow 2^{13}_c \leftarrow 1^{12}_{ac}$	shift d
16	$0^1_\varnothing \leftarrow 2^{12}_a \leftarrow 1^{13}_{ac}$; $0^1_\varnothing \leftarrow 2^{13}_{abc} \leftarrow \mathbf{1^{34}_{abcd}}$; $0^1_\varnothing \leftarrow 2^{13}_c \leftarrow 1^{12}_{ac}$	reduce 1
17	$0^1_\varnothing \leftarrow 2^{12}_a \leftarrow 1^{13}_{ac}$; $0^1_\varnothing \leftarrow 2^{13}_{abc} \leftarrow \mathbf{5^{341}_{abcd}}$; $0^1_\varnothing \leftarrow 2^{13}_c \leftarrow 1^{12}_{ac}$	reduce 3
18	$0^1_\varnothing \leftarrow 2^{12}_a \leftarrow 1^{13}_{ac}$; $0^1_\varnothing \leftarrow \mathbf{2^{14}_{abcd}}$; $0^1_\varnothing \leftarrow 2^{13}_c \leftarrow 1^{12}_{ac}$	accept

$0^1_\varnothing$ 1^{12}_a. (See Sect. 5 for a discussion of strategies.) State $Q_1(1,2)$ just allows a reduction by rule 1, producing a nonterminal edge $\mathsf{G}(1,2)$. This pops 1^{12}_a from the stack; processing $\mathsf{G}(1,2)$ pushes the concrete state $Q_2(1,2)$, which is represented by 2^{12}_a in step 2.

State 5^{231}_{ab} in step 7 allows both, to shift $d = \mathsf{e}(3,4)$, and to reduce by rule 3, with nonterminal edge $\mathsf{G}(1,3)$. The resulting GSS nodes are 1^{34}_{abd} and 2^{13}_{ab} in step 8. State 5^{341}_{cd} allows just a reduce action by grammar rule 3. However, this reduce operation with nonterminal edge $\mathsf{G}(1,4)$ is invalid. If it were valid, $\mathsf{G}(1,4)$ could be derived to the graph consisting of just $c = \mathsf{e}(2,3)$ and $d = \mathsf{e}(3,4)$, i.e., it would generate node 3. However, this contradicts the fact that the unread edge $b = \mathsf{e}(2,3)$ is attached to node 3, which must be generated earlier in the derivation. Therefore, the stack with top-most state 5^{341}_{cd} is discarded in this step (indicated by the asterisk), and analogously in steps 12 and 13.

Note that the shift action in step 9 results in a GSS where 1^{34}_{abd} is the top-most state of two stacks. The reduce operation in step 10, however, removes 1^{34}_{abd} from the GSS and from the corresponding stacks again, and produces the two stacks with top-most states 5^{341}_{abd} and 5^{342}_{abd}.

The GSS in step 18 contains node 2^{14}_{abcd}, i.e., the accept state $Q_2(1,4)$ with the entire input graph being read. The GPSR parser, therefore, has found a successful parse of the input graph.

The current implementation stops when the first successful parse has been found. Another successful parse could have been found if 1^{12}_{ac} had been processed in step 5 or later.

5 Parsing Experiments

We now report on runtime experiments with different parsers applied to series-parallel graphs and to structured flowcharts. The latter are flowcharts that do not allow arbitrary jumps, but represent structured programs with conditional statements and while loops. They consist of rectangles containing instructions, diamonds that indicate conditions, and ovals indicating begin and end of the program. Arrows indicate control flow; see Fig. 2 for an example. Flowcharts are easily represented by graphs as also shown in Fig. 2. Figure 3 defines the rules of an HR grammar generating all graphs representing structured flowcharts. This grammar is not PSR because a state of its CFA has conflicts.

We generated three different parsers for the grammar of series-parallel graphs and for structured flowcharts: a *Cocke-Younger-Kasami* style parser (CYK, [11]) using *DiaGen*[7], and two variants of the GPSR parser using *Grappa* (see footnote 1). The CYK parser was in fact optimized in two ways: the parser creates nonterminal edges by dynamic programming, and each of these edges can be derived to a certain subgraph of the input graph. The optimized parser makes sure that it does not create two or more indistinguishable nonterminals for the same subgraph, even if the nonterminals represent different derivation trees. And it stops as soon as it finds the first derivation of the entire input graph.

[7] Homepage: www.unibw.de/inf2/diagen.

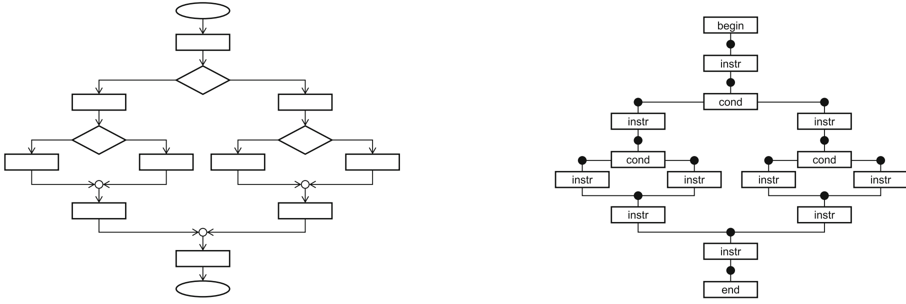


Fig. 2. A structured flowchart (text within the blocks has been omitted) and its graph representation.

$$
\begin{aligned}
\mathsf{Z}() &\to \mathsf{begin}(x)\,\mathsf{P}(x,y)\,\mathsf{end}(y)\\
\mathsf{P}(x,y) &\to \mathsf{S}(x,y) \;\mid\; \mathsf{P}(x,z)\,\mathsf{S}(z,y)\\
\mathsf{S}(x,y) &\to \mathsf{instr}(x,y) \;\mid\\
&\qquad \mathsf{cond}(x,u,v)\,\mathsf{P}(u,y)\,\mathsf{P}(v,y) \;\mid\\
&\qquad \mathsf{cond}(x,u,y)\,\mathsf{P}(u,x)
\end{aligned}
$$

Fig. 3. HR rules for structured flowcharts.

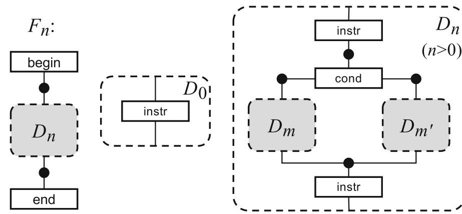
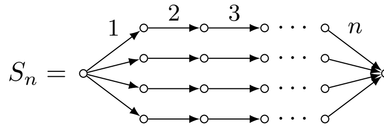


Fig. 4. Definition of flowchart graphs F_n.

The GPSR parsers differ in the strategy that controls which of the currently considered stacks is selected for the next step. GPSR 1 simply maintains a FIFO queue of all such stacks, i.e., new states are enqueued as soon as they are created, and a top-most state is selected for processing as soon as it is next in the queue. GPSR 2, however, applies a more sophisticated strategy. It requires grammar rules to be annotated with either first or second priority. The GPSR 2 parser provides two queues, the first one using FIFO and the second LIFO. New states that result from handling a first priority rule go into the first queue, the others into the second. The parser always tries to select states from the first queue; it selects from the second queue only if the first queue is empty. This way one can control, by annotating grammar rules, which rules should be considered first. This does not affect the correctness of the parser; it can still examine the entire search space. However, it will stop as soon as it finds the first successful parse. By appropriately annotating grammar rules, one can thus speed up the parser if the input graph is valid. However, there is no speed-up for invalid input graphs, since the parser must inspect the entire search space in this case.

The GPSR 2 parser for series-parallel graphs gives rule 3 (series) precedence over rule 2 (parallel); it has been applied to graphs



with different values of n. The GPSR 2 parser for structured flowcharts gives sequences priority over conditional statements; it has been applied to flowcharts F_n defined in Fig. 4 and consisting of n conditions and $3n + 1$ instructions. The flowchart in Fig. 2 is in fact F_3. F_n has a subgraph D_n, which, for $n > 0$, contains subgraphs D_m and $D_{m'}$ with $n = m + m' + 1$. Note that the conditions in F_n form a binary tree with n nodes when we ignore instructions. We always choose m and m' such that it is a complete binary tree.

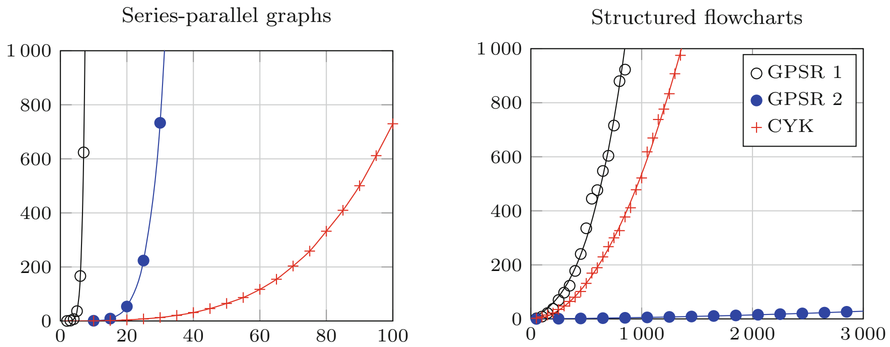


Fig. 5. Runtime (in milliseconds) of different parsers for series-parallel graphs and structured flowcharts.

Figure 5 shows the runtime of the different parsers applied to S_n and F_n with varying value n. Runtime has been measured on an iMac 2017, 4.2 GHz Intel Core i7, Java 1.8.0_181 with standard configuration, and is shown in milliseconds on the y-axis while n is shown on the x-axis.

The experiments first demonstrate that the more sophisticated strategy of GPSR 2 really pays off as GPSR 2 finds a derivation much faster than GPSR 1. For parsing F_{1000}, e.g., GPSR 1 needs 4 013 880 steps, but GPSR 2 just 13 004. The experiments also show that GPSR 1 is in fact much slower than CYK, which demonstrates the need for a sophisticated strategy for the GPSR parser. But for series-parallel graphs, even GPSR 2 is much slower than CYK. Because, the grammar of series-parallel graphs is highly ambiguous. For instance, S_{100} has the ridiculous number of $6.1 \cdot 10^{281}$ derivation trees. The CYK parser has to create 40 422 nonterminal edges for S_{100}, and for S_{40} just 6 582, where most of them represent a high number of different derivation trees. GPSR 2, however, needs 908 122 steps to find a derivation for S_{40}. Apparently, the compactification by the optimized CYK is more effective to cut down the number of choices the parser has to follow.

6 Conclusions

We have generalized PSR parsing for HR grammars [6] to cope with ambiguous graph grammars, by pursuing all possible parses of a graph in parallel until the first derivation has been found. This work is inspired by Tomita's GLR string parsers [12], which extend D.E. Knuth's LR string parsers [9]. For the academic example grammars examined in Sect. 5, in particular the highly ambiguous grammar for series-parallel graphs, comparison of our parser with the CYK parser does not give a clear picture. Moreover, the speed-up obtained by choosing an appropriate strategy only helps when parsing valid graphs, but not when processing invalid graphs. Our experiments shall be extended in two respects: First, we shall study more, and more realistic HR grammars, e.g., the modestly ambiguous (and big) grammars used for processing abstract meaning representations in natural language processing (NLP). Second, we shall compare the GPSR parser with the two parsers used for NLP: the Bolinas parser [1] by D. Chiang, K. Knight *et al.* implements the polynomial algorithm for a restricted class of HR grammars devised in [10]; the s-graph parser [7] by A. Koller *et al.* uses a similar formalism.

We also intend to extend both the original and the generalized PSR parsers to contextual HR grammars [2,3], which have greater generative power, and can be used for analyzing graph models that are more general, and more relevant in practice. Our experience with PTD parsing [4] suggests that this should be relatively easy.

References

1. Chiang, D., Andreas, J., Bauer, D., Hermann, K.M., Jones, B., Knight, K.: Parsing graphs with hyperedge replacement grammars. In: Proceedings of the 51st Annual Meeting Association for Computational Linguistics, vol. 1, pp. 924–932 (2013)
2. Drewes, F., Hoffmann, B.: Contextual hyperedge replacement. Acta Inf. **52**, 497–524 (2015)
3. Drewes, F., Hoffmann, B., Minas, M.: Contextual hyperedge replacement. In: Schürr, A., Varró, D., Varró, G. (eds.) AGTIVE 2011. LNCS, vol. 7233, pp. 182–197. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34176-2_16
4. Drewes, F., Hoffmann, B., Minas, M.: Predictive top-down parsing for hyperedge replacement grammars. In: Parisi-Presicce, F., Westfechtel, B. (eds.) ICGT 2015. LNCS, vol. 9151, pp. 19–34. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21145-9_2
5. Drewes, F., Hoffmann, B., Minas, M.: Approximating Parikh images for generating deterministic graph parsers. In: Milazzo, P., Varró, D., Wimmer, M. (eds.) STAF 2016. LNCS, vol. 9946, pp. 112–128. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-50230-4_9
6. Drewes, F., Hoffmann, B., Minas, M.: Formalization and correctness of predictive shift-reduce parsers for graph grammars based on hyperedge replacement. J. Log. Algebr. Meth. Program. (2018). https://doi.org/10.1016/j.jlamp.2018.12.006. https://arxiv.org/abs/1812.11927

7. Groschwitz, J., Koller, A., Teichmann, C.: Graph parsing with s-graph grammars. In: Proceedings of the 53rd Annual Meeting Association for Computational Linguistics, ACL 2015, Volume 1: Long Papers, pp. 1481–1490. The Association for Computer Linguistics (2015)
8. Habel, A.: Hyperedge Replacement: Grammars and Languages. LNCS, vol. 643. Springer, Heidelberg (1992). https://doi.org/10.1007/BFb0013875
9. Knuth, D.E.: On the translation of languages from left to right. Inf. Control **8**(6), 607–639 (1965)
10. Lautemann, C.: The complexity of graph languages generated by hyperedge replacement. Acta Inf. **27**, 399–421 (1990)
11. Minas, M.: Concepts and realization of a diagram editor generator based on hypergraph transformation. Sci. Comput. Program. **44**(2), 157–180 (2002)
12. Tomita, M.: An efficient context-free parsing algorithm for natural languages. In: Proceedings of the 9th International Joint Conference on Artificial Intelligence, pp. 756–764. Morgan Kaufmann (1985)