Graph parsing with s-graph grammars

Jonas Groschwitz and **Alexander Koller** and **Christoph Teichmann**
Department of Linguistics
University of Potsdam
`firstname.lastname@uni-potsdam.de`

Abstract

A key problem in semantic parsing with graph-based semantic representations is *graph parsing*, i.e. computing all possible analyses of a given graph according to a grammar. This problem arises in training synchronous string-to-graph grammars, and when generating strings from them. We present two algorithms for graph parsing (bottom-up and top-down) with s-graph grammars. On the related problem of graph parsing with hyperedge replacement grammars, our implementations outperform the best previous system by several orders of magnitude.

1 Introduction

The recent years have seen an increased interest in *semantic parsing*, the problem of deriving a semantic representation for natural-language expressions with data-driven methods. With the recent availability of graph-based meaning banks (Banarescu et al., 2013; Oepen et al., 2014), much work has focused on computing graph-based semantic representations from strings (Jones et al., 2012; Flanigan et al., 2014; Martins and Almeida, 2014).

One major approach to graph-based semantic parsing is to learn an explicit synchronous grammar which relates strings with graphs. One can then apply methods from statistical parsing to parse the string and read off the graph. Chiang et al. (2013) and Quernheim and Knight (2012) represent this mapping of a (latent) syntactic structure to a graph with a grammar formalism called *hyperedge replacement grammar* (HRG; (Drewes et al., 1997)). As an alternative to HRG, Koller (2015) introduced *s-graph grammars* and showed that they support linguistically reasonable grammars for graph-based semantics construction.

One problem that is only partially understood in the context of semantic parsing with explicit grammars is *graph parsing*, i.e. the computation of the possible analyses the grammar assigns to an input *graph* (as opposed to string). This problem arises whenever one tries to generate a string from a graph (e.g., on the generation side of an MT system), but also in the context of extracting and training a synchronous grammar, e.g. in EM training. The state of the art is defined by the bottom-up graph parsing algorithm for HRG by Chiang et al. (2013), implemented in the Bolinas tool (Andreas et al., 2013).

We present two graph parsing algorithms (top-down and bottom-up) for s-graph grammars. S-graph grammars are equivalent to HRGs, but employ a more fine-grained perspective on graph-combining operations. This simplifies the parsing algorithms, and facilitates reasoning about them. Our bottom-up algorithm is similar to Chiang et al.'s, and derives the same asymptotic number of rule instances. The top-down algorithm is novel, and achieves the same asymptotic runtime as the bottom-up algorithm by reasoning about the biconnected components of the graph. Our evaluation on the "Little Prince" graph-bank shows that our implementations of both algorithms outperform Bolinas by several orders of magnitude. Furthermore, the top-down algorithm can be more memory-efficient in practice.

2 Related work

The AMR-Bank (Banarescu et al., 2013) annotates sentences with *abstract meaning representations (AMRs)*, like the one shown in Fig. 1(a). These are graphs that represent the predicate-argument structure of a sentence; notably, phenomena such as control are represented by reentrancies in the graph. Another major graph-bank is the SemEval-2014 shared task on semantic dependency parsing dataset (Oepen et al., 2014).

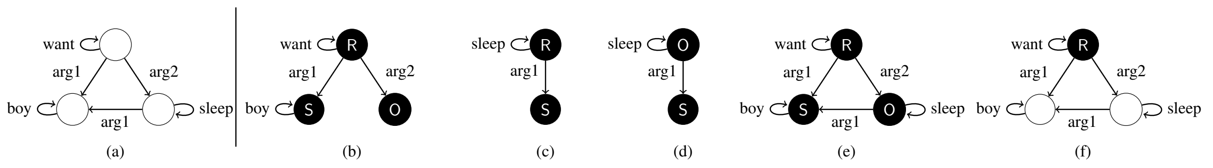

Figure 1: AMR (a) for 'The boy wants to sleep', and s-graphs. We call (b) SG_{want} and (c) SG_{sleep}.

The primary grammar formalism currently in use for synchronous graph grammars is *hyperedge replacement grammar (HRG)* (Drewes et al., 1997), which we sketch in Section 4.3. An alternative is offered by Koller (2015), who introduced *s-graph grammars* and showed that they lend themselves to manually written grammars for semantic construction. In this paper, we show the equivalence of HRG and s-graph grammars and work out graph parsing for s-graph grammars.

The first polynomial graph parsing algorithm for HRGs on graphs with limited connectivity was presented by Lautemann (1988). Lautemann's original algorithm is a top-down parser, which is presented at a rather abstract level that does not directly support implementation or detailed complexity analysis. We extend Lautemann's work by showing how new parse items can be represented and constructed efficiently. Finally, Chiang et al. (2013) presented a bottom-up graph parser for HRGs, in which the representation and construction of items was worked out for the first time. It produces $O((n \cdot 3^d)^{k+1})$ instances of the rules in a parsing schema, where n is the number of nodes of the graph, d is the maximum degree of any node, and k is a quantity called the *tree-width* of the grammar.

3 An algebra of graphs

We start by introducing the exact type of graphs that our grammars and parsers manipulate, and by developing some theory.

Throughout this paper, we define a *graph* $G = (V, E)$ as a directed graph with edge labels from some label alphabet L. The graph consists of a finite set V of nodes and a finite set $E \subseteq V \times V \times L$ of edges $e = (u, v, l)$, where u and v are the nodes connected by e, and $l \in L$ is the *edge label*. We say that e is *incident* to both u and v, and call the number of edges incident to a node its *degree*. We write $u \overset{e}{\leftrightarrow} v$ if either $e = (u, v, l)$ or $e = (v, u, l)$ for some l; we drop the e if the identity of the edge is irrelevant. Edges with $u = v$ are called *loops*; we use them here to encode node labels. Given a graph G, we write $n = |V|$, $m = |E|$, and d for the maximum degree of any node in V.

If $f : A \rightsquigarrow B$ and $g : A \rightsquigarrow B$ are partial functions, we let the partial function $f \cup g$ be defined if for all $a \in A$ with both $f(a)$ and $g(a)$ defined, we have $f(a) = g(a)$. We then let $(f \cup g)(a)$ be $f(a)$ if $f(a)$ is defined; $g(a)$ if $g(a)$ is defined; and undefined otherwise.

3.1 The HR algebra of graphs with sources

Our grammars describe how to build graphs from smaller pieces. They do this by accessing nodes (called *source nodes*) which are assigned "public names". We define an *s-graph* (Courcelle and Engelfriet, 2012) as a pair $SG = (G, \phi)$ of a graph G and a *source assignment*, i.e. a partial, injective function $\phi : S \rightsquigarrow V$ that maps some *source names* from a finite set S to the nodes of G. We call the nodes in $\phi(S)$ the *source nodes* or *sources* of SG; all other nodes are *internal nodes*. If ϕ is defined on the source name σ, we call $\phi(\sigma)$ the *σ-source* of SG. Throughout, we let $s = |S|$.

Examples of s-graphs are given in Fig. 1. We use numbers as node names and lowercase strings for edge names (except in the concrete graphs of Fig. 1, where the edges are marked with edge labels instead). Source nodes are drawn in black, with source names drawn on the inside. Fig. 1(b) shows an s-graph SG_{want} with three nodes and four edges. The three nodes are marked as the R-, S-, and O-source, respectively. Likewise, the s-graph SG_{sleep} in (c) has two nodes (one of which is an R-source and the other an S-source) and two edges.

We can now apply operations to these graphs. First, we can *rename* the R-source of (c) to an O-source. The result, denoted $SG_d = SG_{sleep}[\mathsf{R} \rightarrow \mathsf{O}]$, is shown in (d). Next, we can *merge* SG_d with SG_{want}. This copies the edges and nodes of SG_d and SG_{want} into a new s-graph; but crucially, for every source name σ the two s-graphs have in common, the σ-sources of the graphs are fused into a single node (and become a σ-source of the result). We write $||$ for the merge operation;

thus we obtain $SG_e = SG_d \mid\mid SG_{want}$, shown in (e). Finally, we can *forget* source names. The graph $SG_f = \mathsf{f_S}(\mathsf{f_O}(SG_e))$, in which we forgot S and O, is shown in (f). We refer to Courcelle and Engelfriet (2012) for technical details.[1]

We can take the set of all s-graphs, together with these operations, as an *algebra* of s-graphs. In addition to the binary merge operation and the unary operations for forget and rename, we fix some finite set of *atomic* s-graphs and take them as constants of the algebra which evaluate to themselves. Following Courcelle and Engelfriet, we call this algebra the *HR algebra*. We can *evaluate* any term τ consisting of these operation symbols into an s-graph $[\![\tau]\!]$ as usual. For instance, the following term encodes the merge, forget, and rename operations from the example above, and evaluates to the s-graph in Fig. 1(f).

(1) $\mathsf{f_S}(\mathsf{f_O}(SG_{want} \mid\mid SG_{sleep}[\mathsf{R} \to \mathsf{O}]))$

The set of s-graphs that can be represented as the value $[\![\tau]\!]$ of some term τ over the HR algebra depends on the source set S and on the constants. For simplicity, we assume here that we have a constant for each s-graph consisting of a single labeled edge (or loop), and that the values of all other constants can be expressed by combining these using merge, rename, and forget.

3.2 S-components

A central question in graph parsing is how some s-graph that is a subgraph of a larger s-graph SG (a *sub-s-graph*) can be represented as the merge of two smaller sub-s-graphs of SG. In general, $SG_1 \mid\mid SG_2$ is defined for any two s-graphs SG_1 and SG_2. However, if we see SG_1 and SG_2 as subgraphs of SG, $SG_1 \mid\mid SG_2$ may no longer be a subgraph of SG. For instance, we cannot merge the s-graphs (b) and (c) in Fig. 2 as part of the graph (a): The startpoints of the edges a and d are both A-sources and would thus become the same node (unlike in (a)), and furthermore the edge d would have to be duplicated. In graph parsing, we already know the identity of all nodes and edges in sub-s-graphs (as nodes and edges in SG), and must thus pay attention that merge operations do not accidentally fuse or duplicate them. In particular, two sub-s-graphs cannot be merged if they have edges in common.

[1] Note that the rename operation of Courcelle and Engelfriet (2012) allows for swapping source assignments and making multiple renames in one step. We simplify the presentation here, but all of our techniques extend easily.

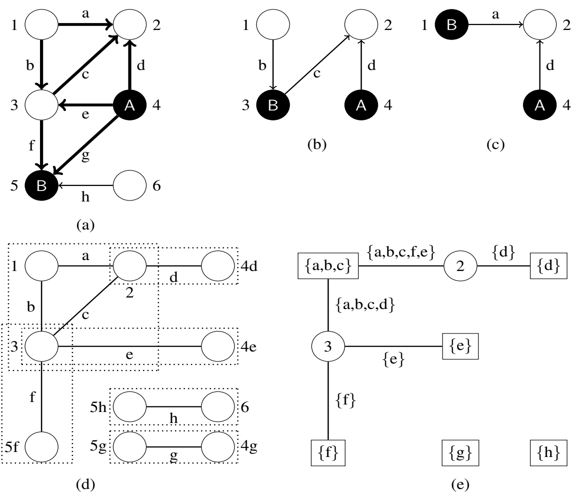

Figure 2: (a) An s-graph with (b,c) some sub-s-graphs, (d) its BCCs, and (e) its block-cutpoint graph.

We call a sub-s-graph SG_1 of SG *extensible* if there is another sub-s-graph SG_2 of SG such that $SG_1 \mid\mid SG_2$ contains the same edges as SG. An example of a sub-s-graph that is not extensible is the sub-s-graph (b) of the s-graph in (a) in Fig. 2. Because sources can only be renamed or forgotten by the algebra operations, but never introduced, we can never attach the missing edge a: this can only happen when 1 and 2 are sources. As a general rule, a sub-s-graph can only be extensible if it contains all edges that are adjacent to all of its internal nodes in SG. Obviously, a graph parser need only concern itself with sub-s-graphs that are extensible.

We can further clarify the structure of extensible sub-s-graphs by looking at the *s-components* of a graph. Let $U \subseteq V$ be some set of nodes. This set splits the edges of G into equivalence classes that are separated by U. We say that two edges $e, f \in E$ are *equivalent* with respect to U, $e \sim_U f$, if there is a sequence $v_1 \overset{e}{\leftrightarrow} v_2 \leftrightarrow \dots v_{k-1} \overset{f}{\leftrightarrow} v_k$ with $v_2, \dots, v_{k-1} \notin U$, i.e. if we can reach f from an endpoint of e without visiting a node in U. We call the equivalence classes of E with respect to $\sim_U$ the *s-components* of G and denote the s-component that contains an edge e with $[e]$. In Fig. 2(a), the edges a and f are equivalent with respect to $U = \{4, 5\}$, but a and h are not. The s-components are $[a] = \{a, b, c, d, e, f\}$, $[g] = \{g\}$, and $[h] = \{h\}$.

It can be shown that for any s-graph $SG =$

(G, ϕ), a sub-s-graph SH with source nodes U is extensible iff its edge set is the union of a set of s-components of G with respect to U. We let an *s-component representation* $\mathcal{C} = (C, \phi)$ in the s-graph $SG = (G, \phi')$ consist of a source assignment $\phi : S \rightsquigarrow V$ and a set C of s-components of G with respect to the set $\mathrm{VS}_{\mathcal{C}} = \phi(S) \subseteq V$ of source nodes of ϕ. Then we can represent every extensible sub-s-graph $SH = (H, \phi)$ of SG by the s-component representation $\mathcal{C} = (C, \phi)$ where C is the set of s-components of which SH consists. Conversely, we write $T(\mathcal{C})$ for the unique extensible sub-s-graph of SG represented by the s-component representation $\mathcal{C}$.

The utility of s-component representations derives from the fact that merge can be evaluated on these representations alone, as follows.

Lemma 1. *Let $\mathcal{C} = (C, \phi), \mathcal{C}_1 = (C_1, \phi_1), \mathcal{C}_2 = (C_2, \phi_2)$ be s-component representations in the s-graph SG. Then $T(\mathcal{C}) = T(\mathcal{C}_1) \mid\mid T(\mathcal{C}_2)$ iff $C = C_1 \uplus C_2$ (i.e., disjoint union) and $\phi_1 \cup \phi_2$ is defined, injective, and equal to ϕ.*

3.3 Boundary representations

If there is no $\mathcal{C}$ such that all conditions of Lemma 1 are satisfied, then $T(\mathcal{C}_1) \mid\mid T(\mathcal{C}_2)$ is not defined. In order to check this efficiently in the bottom-up parser, it will be useful to represent s-components explicitly via their *boundary*.

Consider an s-component representation $\mathcal{C} = (C, \phi)$ in SG and let E be the set of all edges that are adjacent to a source node in $\mathrm{VS}_{\mathcal{C}}$ and contained in an s-component in C. Then we let the *boundary representation (BR)* β of $\mathcal{C}$ in the s-graph SG be the pair $\beta = (E, \phi)$. That is, β represents the s-components through the *in-boundary edges*, i.e. those edges inside the s-components (and thus the sub-s-graph) which are adjacent to a source. The BR β specifies $\mathcal{C}$ uniquely if the base graph SG is connected, so we write $T(\beta)$ for $T(\mathcal{C})$ and VS_β for $\mathrm{VS}_{\mathcal{C}}$.

In Fig. 2(a), the bold sub-s-graph is represented by $\beta = \langle \{d, e, f, g\}, \{\mathsf{A}{:}4, \mathsf{B}{:}5\} \rangle$, indicating that it contains the A-source 4 and the B-source 5; and further, that the edge set of the sub-s-graph is $[d] \cup [e] \cup [f] \cup [g] = \{a, b, c, d, e, f, g\}$. The edge h (which is also incident to 5) is not specified, and therefore not in the sub-s-graph.

The following lemma can be shown about computing merge on boundary representations. Intuitively, the conditions (b) and (c) guarantee that the component sets are disjoint; the lemma then follows from Lemma 1.

Lemma 2. *Let SG be an s-graph, and let $\beta_1 = (E_1, \phi_1), \beta_2 = (E_2, \phi_2)$ be two boundary representations in SG. Then $T(\beta_1) \mid\mid T(\beta_2)$ is defined within SG iff the following conditions hold:*

(a) $\phi_1 \cup \phi_2$ is defined and injective;

(b) the two BRs have no in-boundary edges in common, i.e. $E_1 \cap E_2 = \emptyset$;

(c) for every source node v of β_1, the last edge on the path in SG from v to the closest source node of β_2 is not an in-boundary edge of β_2, and vice versa.

Furthermore, if these conditions hold, we have $T(\beta_1 \mid\mid \beta_2) = T(\beta_1) \mid\mid T(\beta_2)$, where we define $\beta_1 \mid\mid \beta_2 = (E_1 \cup E_2, \phi_1 \cup \phi_2)$.

4 S-graph grammars

We are now ready to define s-graph grammars, which describe languages of s-graphs. We also introduce graph parsing and relate s-graph grammars to HRGs.

4.1 Grammars for languages of s-graphs

We use *interpreted regular tree grammars* (IRTGs; Koller and Kuhlmann (2011)) to describe languages of s-graphs. IRTGs are a very general mechanism for describing languages over and relations between arbitrary algebras. They separate conceptually the generation of a grammatical derivation from its interpretation as a string, tree, graph, or some other object.

Consider, as an example, the tiny grammar in Fig. 3; see Koller (2015) for linguistically meaningful grammars. The left column consists of a regular tree grammar $\mathcal{G}$ (RTG; see e.g. Comon et al. (2008)) with two rules. This RTG describes a regular language $L(\mathcal{G})$ of *derivation trees* (in general, it may be infinite). In the example, we can derive $\mathsf{S} \Rightarrow \mathsf{r}_1(\mathsf{VP}) \Rightarrow \mathsf{r}_1(\mathsf{r}_2)$, therefore we have $t = \mathsf{r}_1(\mathsf{r}_2) \in L(\mathcal{G})$.

We then use a *tree homomorphism* h to rewrite the derivation trees into terms over an algebra; in this case the HR algebra. In the example, the values $h(\mathsf{r}_1)$ and $h(\mathsf{r}_2)$ are specified in the second column of Fig. 3. We compute $h(t)$ by substituting the variable x_1 in $h(\mathsf{r}_1)$ with $h(\mathsf{r}_2)$. The term $h(t)$ is thus the one shown in (1). It evaluates to the s-graph SG_f in Fig. 1(f).

Rule of RTG $\mathcal{G}$	homomorphism h
$\mathsf{S} \to \mathsf{r}_1(\mathsf{VP})$	$\mathsf{f_S}(\mathsf{f_O}(SG_{want} \mid\mid x_1[\mathsf{R} \to \mathsf{O}]))$
$\mathsf{VP} \to \mathsf{r}_2$	SG_{sleep}

Figure 3: An example s-graph grammar.

In general, the IRTG $\mathbb{G} = (\mathcal{G}, h, \mathcal{A})$ generates the *language* $L(\mathbb{G}) = \{[\![h(t)]\!] \mid t \in L(\mathcal{G})\}$, where $[\![\cdot]\!]$ is evaluation in the algebra $\mathcal{A}$. Thus, in the example, we have $L(\mathbb{G}) = \{SG_f\}$.

In this paper, we focus on IRTGs that describe languages $L(\mathbb{G}) \subseteq \mathcal{A}$ of objects in an algebra; specifically, of s-graphs in the HR algebra. However, IRTGs extend naturally to a synchronous grammar formalism by adding more homomorphisms and algebras. For instance, the grammars in Koller (2015) map each derivation tree simultaneously to a string and an s-graph, and therefore describe a binary relation between strings and s-graphs. We call IRTGs where at least one algebra is the HR algebra, *s-graph grammars*.

4.2 Parsing with s-graph grammars

In this paper, we are concerned with the parsing problem of s-graph grammars. In the context of IRTGs, parsing means that we are looking for those derivation trees t that are (a) grammatically correct, i.e. $t \in L(\mathcal{G})$, and (b) match some given input object a, i.e. $h(t)$ evaluates to a in the algebra. Because the set P of such derivation trees may be large or infinite, we aim to compute an RTG $\mathcal{G}_a$ such that $L(\mathcal{G}_a) = P$. This RTG plays the role of a parse chart, which represents the possible derivation trees compactly.

In order to compute $\mathcal{G}_a$, we need to solve two problems. First, we need to determine all the possible ways in which a can be represented by terms τ over the algebra $\mathcal{A}$. This is familiar from string parsing, where a CKY parse chart spells out all the ways in which larger substrings can be decomposed into smaller parts by concatenation. Second, we need to identify all those derivation trees $t \in L(\mathcal{G})$ that map to such a decomposition τ, i.e. for which $h(t)$ evaluates to a. In string parsing, this corresponds to retaining only such decompositions into substrings that are justified by the grammar rules.

While any parsing algorithm must address both of these issues, they are usually conflated, in that parse items combine information about the decomposition of a (such as a string span) with information about grammaticality (such as nonterminal symbols). In IRTG parsing, we take a different, more generic approach. We assume that the set D of all decompositions τ, i.e. of all terms τ that evaluate to a in the algebra, can be represented as the language $D = L(D_a)$ of a *decomposition grammar* D_a. D_a is an RTG over the signature of the algebra. Crucially, D_a only depends on the algebra and a itself, and not on $\mathcal{G}$ or h, because D contains *all* terms that evaluate to a and not just those that are licensed by the grammar. However, we can compute $\mathcal{G}_a$ from D_a efficiently by exploiting the closure of regular tree languages under intersection and inverse homomorphism; see Koller and Kuhlmann (2011) for details.

In practice, this means that whenever we want to apply IRTGs to a new algebra (as, in this paper, to the HR algebra), we can obtain a parsing algorithm by specifying how to compute decomposition grammars over this algebra. This is the topic of Section 5.

4.3 Relationship to HRG

We close our exposition of s-graph grammars by relating them to HRGs. It is known that the graph languages that can be described with s-graph grammars are the same as the HRG languages (Courcelle and Engelfriet, 2012, Prop. 4.27). Here we establish a more precise equivalence result, so we can compare our asymptotic runtimes directly to those of HRG parsers.

An HRG rule, such as the one shown in Fig. 4, rewrites a nonterminal symbol into a graph. The example rule constructs a graph for the nonterminal S by combining the graph G_r in the middle (with nodes $1, 2, 3$ and edges e, f) with graphs G_X and G_Y that are recursively derived from the nonterminals X and Y. The combination happens by merging the *external* nodes of G_X and G_Y with nodes of G_r: the squiggly lines indicate that the external node I of G_X should be 1, and the external node II should be 2. Similarly the external nodes of G_Y are unified with 1 and 3. Finally, the external nodes I and II of the HRG rule for S itself, shaded gray, are 1 and 3.

The fundamental idea of the HRG-to-IRTG translation is to encode external nodes as sources, and to use rename and merge to unify the nodes of the different graphs. In the example, we might say that the external nodes of G_X and G_Y are represented using the source names I and II, and extend G_r to an s-graph by saying that the nodes 1, 2, and

3 are its I-source, III-source, and II-source respectively. This results in the expression

(2) $\mathsf{f}_{\text{III}}(\langle\text{I}\rangle \xrightarrow{e} \langle\text{III}\rangle \mid\mid x_1[\text{II} \to \text{III}]$
$\mid\mid \langle\text{I}\rangle \xrightarrow{f} \langle\text{II}\rangle \mid\mid x_2)$

where we write "$\langle\text{I}\rangle \xrightarrow{e} \langle\text{III}\rangle$" for the s-graph consisting of the edge e, with node 1 as I-source and 2 as III-source.

However, this requires the use of three source names (I, II, and III). The following encoding of the rule uses the sources more economically:

(3) $\mathsf{f}_{\text{II}}(\langle\text{I}\rangle \xrightarrow{e} \langle\text{II}\rangle \mid\mid x_1) \mid\mid \langle\text{I}\rangle \xrightarrow{f} \langle\text{II}\rangle \mid\mid x_2$

This term uses only two source names. It forgets II as soon as we are finished with the node 2, and frees the name up for reuse for 3. The complete encoding of the HRG rule consists of the RTG rule S $\to$ r(X, Y) with $h(\mathsf{r}) = (3)$.

In the general case, one can "read off" possible term encodings of a HRG rule from its *tree decompositions*; see Chiang et al. (2013) or Def. 2.80 of Courcelle and Engelfriet (2012) for details. A tree decomposition is a tree, each of whose nodes π is labeled with a subset V_π of the nodes in the HRG rule. We can construct a term encoding from a tree decomposition bottom-up. Leaves map to variables or constants; binary nodes introduce merge operations; and we use rename and forget operations to ensure that the subterm for the node π evaluates to an s-graph in which exactly the nodes in V_π are source nodes.[2] In the example, we obtain (3) from the tree decomposition in Fig. 4 like this.

The *tree-width* k of an HRG rule is measured by finding the tree decomposition of the rule for which the node sets have the lowest maximum size s and setting $k = s - 1$. It is a crucial measure because Chiang et al.'s parsing algorithm is exponential in k. The translation we just sketched uses s source names. Thus we see that a HRG with rules of tree-width $\leq k$ can be encoded into an s-graph grammar with $k + 1$ source names. (The converse is also true.)

5 Graph parsing with s-graph grammars

Now we show how to compute decomposition grammars for the s-graph algebra. As we explained in Section 4.2, we can then obtain a complete parser for s-graph grammars through generic methods.

[2]This uses the swap operations mentioned in Footnote 1.

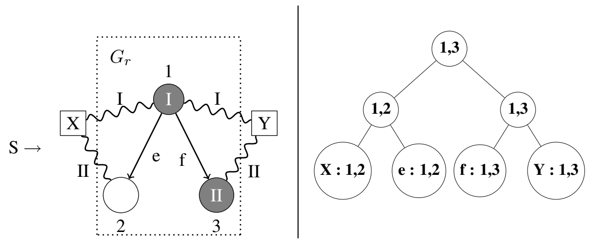

Figure 4: An HRG rule (left) with one of its tree decompositions (right).

Given an s-graph SG, the language of the decomposition grammar D_{SG} is the set of all terms over the HR algebra that evaluate to SG. For example, the decomposition grammar for the graph SG in Fig. 1(a) contains – among many others – the following two rules:

(4) $SG \to \mathsf{f}_{\mathsf{R}}(SG_f)$
(5) $SG_e \to \mid\mid (SG_b, SG_d)$,

where SG_f, SG_e, SG_b, and SG_d are the graphs from Fig. 1 (see Section 3.1). In other words, D_{SG} keeps track of sub-s-graphs in the nonterminals, and the rules spell out how "larger" sub-s-graphs can be constructed from "smaller" sub-s-graphs using the operations of the HR algebra. The algorithms below represent sub-s-graphs compactly using s-component and boundary representations.

Because the decomposition grammars in the s-graph algebra can be very large (see Section 6), we will not usually compute the entire decomposition grammar explicitly. Instead, it is sufficient to maintain a lazy representation of D_{SG}, which allows us to answer *queries* to the decomposition grammar efficiently. During parsing, such queries will be generated by the generic part of the parsing algorithm. Specifically, we will show how to answer the following types of query:

- *Top-down:* given an s-component representation $\mathcal{C}$ of some s-graph and an algebra operation o, enumerate all the rules $\mathcal{C} \to o(\mathcal{C}_1, \ldots, \mathcal{C}_k)$ in D_{SG}. This asks how a larger sub-s-graph can be derived from other sub-s-graphs using the operation o. In the example above, a query for SG and $\mathsf{f}_{\mathsf{R}}(\cdot)$ should yield, among others, the rule in (4).

- *Bottom-up:* given boundary representations $\beta_1, \ldots, \beta_k$ and an algebra operation o, enumerate all the rules $\beta \to o(\beta_1, \ldots, \beta_k)$ in D_{SG}. This asks how smaller sub-s-graphs can be combined into a bigger one using the

	forget	rename	merge
bottom-up	$O(d+s)$	$O(s)$	$O(ds)$
top-down	$O(ds)$	$O(s)$	$O(ds)$
$I =$ # rules	$O(n^s 2^{ds})$	$O(n^s 2^{ds})$	$O(n^s 3^{ds})$

Table 1: Amortized per-rule runtimes T for the different rule types.

operation o. In the example above, a merge query for SG_b and SG_d should yield the rule in (5). Unlike in the top-down case, every bottom-up query returns at most one rule.

The runtime of the complete parsing algorithm is bounded by the number I of different queries to D_{SG} that we receive, multiplied by the *per-rule runtime* T that we need to answer each query. The factor I is analogous to the number of rule instances in schema-based parsing (Shieber et al., 1995). The factor T is often ignored in the analysis of parsing algorithms, because in parsing schemata for strings, we typically have $T = O(1)$. This need not be the case for graph parsers. In the HRG parsing schema of Chiang et al. (2013), we have $I = O(n^{k+1} 3^{d(k+1)})$, where k is the treewidth of the HRG. In addition, each of their rule instances takes time $T = O(d(k+1))$ to actually calculate the new item.

Below, we show how we can efficiently answer both bottom-up and top-down queries to D_{SG}. Every s-graph grammar has an equivalent normal form where every constant describes an s-graph with a single edge. Assuming that the grammar is in this normal form, queries of the form $\beta \rightarrow g$ (resp. $\mathcal{C} \rightarrow g$), where g is a constant of the HR-algebra, are trivial and we will not consider them further. Table 1 summarizes our results.

5.1 Bottom-up decomposition

Forget and rename. Given a boundary representation $\beta' = (E', \phi')$, answering the bottom-up forget query $\beta \rightarrow \mathsf{f_A}(\beta')$ amounts to verifying that all edges incident to $\phi'(\mathsf{A})$ are in-boundary in β', since otherwise the result would not be extensible. This takes time $O(d)$. We then let $\beta = (E, \phi)$, where ϕ is like ϕ' but undefined on A, and E is the set of edges in E' that are still incident to a source in ϕ. Computing β thus takes time $O(d+s)$.

The rename operation works similarly, but since the edge set remains unmodified, the per-rule runtime is $O(s)$.

A BR is fully determined by specifying the node and in-boundary edges for each source name, so there are at most $O\left(\left(n2^d\right)^s\right)$ different BRs. Since the result of a forget or rename rule is determined by the child β', this is an upper bound for the number I of rule instances of forget or rename.

Merge. Now consider the bottom-up merge query for the boundary representations β_1 and β_2. As we saw in Section 3.3, $T(\beta_1) \mid\mid T(\beta_2)$ is not always defined. But if it is, we can answer the query with the rule $(\beta_1 \mid\mid \beta_2) \rightarrow \; \mid\mid (\beta_1, \beta_2)$, with $\beta_1 \mid\mid \beta_2$ defined as in Section 3.3. Computing this BR takes time $O(ds)$.

We can check whether $T(\beta_1) \mid\mid T(\beta_2)$ is defined by going through the conditions of Lemma 2. The only nontrivial condition is (c). In order to check it efficiently, we precompute a data structure which contains, for any two nodes $u, v \in V$, the length k of the shortest undirected path $u = v_1 \leftrightarrow \ldots \overset{e}{\leftrightarrow} v_k = v$ and the last edge e on this path. This can be done in time $O(n^3)$ using the Floyd-Warshall algorithm. Checking (c) for every source pair then takes time $O(s^2)$ per rule, but because sources that are common to both β_1 and β_2 automatically satisfy (c) due to (a), one can show that the total runtime of checking (c) for all merge rules of $D_S G$ is $O(n^s 3^{ds} s)$.

Observe finally that there are $I = O(n^s 3^{ds})$ instances of the merge rule, because each of the $O(ds)$ edges that are incident to a source node can be either in β_1, in β_2, or in neither. Therefore the runtime for checking (c) amortizes to $O(s)$ per rule. The Floyd-Warshall step amortizes to $O(1)$ per rule for $s \geq 3$; for $s \leq 2$ the node table can be computed in amortized $O(1)$ using more specialized algorithms. This yields a total amortized per-rule runtime T for bottom-up merge of $O(ds)$.

5.2 Top-down decomposition

For the top-down queries, we specify sub-s-graphs in terms of their s-component representations. The number I of instances of each rule type is the same as in the bottom-up case because of the one-to-one correspondence of s-component and boundary representations. We focus on merge and forget queries; rename is as above.

Merge. Given an s-component representation $\mathcal{C} = (C, \phi)$, a top-down merge query asks us to enumerate the rules $\mathcal{C} \rightarrow \; \mid\mid (\mathcal{C}_1, \mathcal{C}_2)$ such that $T(\mathcal{C}_1) \mid\mid T(\mathcal{C}_2) = T(\mathcal{C})$. By Lemma 1, we can do this by using every distribution of the s-components in C over $\mathcal{C}_1$ and $\mathcal{C}_2$ and restricting ϕ

accordingly. This brings the per-rule time of top-down merge to $O(ds)$, the maximum number of s-components in C.

Block-cutpoint graphs. The challenging query to answer top-down is forget. We will first describe the problem and introduce a data structure that supports efficient top-down forget queries.

Consider top-down forget queries on the sub-s-graph SG_1 drawn in bold in Fig. 2(a); its s-component representation is $\langle\{[a],[g]\},\{\mathsf{A}{:}4,\mathsf{B}{:}5\}\rangle$. A top-down forget might promote the node 3 to a C-source, yielding a sub-s-graph SG_2 (that is, $\mathsf{f}_\mathsf{C}(SG_2)$ is the original s-graph SG_1). In SG_2, a, e, and f are no longer equivalent; its s-component representation is $\langle\{[a],[e],[f],[g]\},\{\mathsf{A}{:}4,\mathsf{B}{:}5,\mathsf{C}{:}3\}\rangle$. Thus promoting 3 to a source splits the original s-component into smaller parts.

By contrast, the same top-down forget might also promote the node 1 to a C-source, yielding a sub-s-graph SG_3; $\mathsf{f}_\mathsf{C}(SG_3)$ is also SG_1. However, all edges in $[a]$ are still equivalent in SG_3; its s-component representation is $\langle\{[a],[g]\},\{\mathsf{A}{:}4,\mathsf{B}{:}5,\mathsf{C}{:}1\}\rangle$.

An algorithm for top-down forget must be able to determine whether promotion of a node splits an s-component or not. To do this, let G be the input graph. We create an undirected auxiliary graph G^U from G and a set U of (source) nodes. G^U contains all nodes in $V \backslash U$, and for each edge e that is incident to a node $u \in U$, it contains a node (u, e). Furthermore, G^U contains undirected versions of all edges in G; if an edge $e \in E$ is incident to a node $u \in U$, it becomes incident to (u, e) in G^U instead. The auxiliary graph $G^{\{4,5\}}$ for our example graph is shown in Fig. 2(d).

Two edges are connected in G^U if and only if they are equivalent with respect to U in G. Therefore, promotion of u splits s-components iff u is a *cutpoint* in G^U, i.e. a node whose removal disconnects the graph. Cutpoints can be characterized as those nodes that belong to multiple *biconnected components (BCCs)* of G^U, i.e. the maximal subgraphs such that any node can be removed without disconnecting a graph segment. In Fig. 2(d), the BCCs are indicated by the dotted boxes. Observe that 3 is a cutpoint and 1 is not.

For any given U, we can represent the structure of the BCCs of G^U in its *block-cutpoint graph*. This is a bipartite graph whose nodes are the cutpoints and BCCs of G^U, and a BCC is connected to all of its cutpoints; see Fig. 2(e) for the block-cutpoint graph of the example. Block-cutpoint graphs are always forests, with the individual trees representing the s-components of G. Promoting a cutpoint u splits the s-component into smaller parts, each corresponding to an incident edge of u. We annotate each edge with that part.

Forget. We can now answer a top-down forget query $\mathcal{C} \rightarrow \mathsf{f}_\mathsf{A}(\mathcal{C}')$ efficiently from the block-cutpoint graph for the sources of $\mathcal{C} = (C, \phi)$. We iterate over all components $c \in C$, and then over all internal nodes u of c. If u is not a cutpoint, we simply let $\mathcal{C}' = (C', \phi')$ by making u an A-source and letting $C' = C$. Otherwise, we also remove c from C and add the new s-components on the edges adjacent to u in the block-cutpoint graph. The query returns rules for all $\mathcal{C}'$ that can be constructed like this.

The per-rule runtime of top-down forget is $O(ds)$, the time needed to compute C' in the cutpoint case. We furthermore precompute the block-cutpoint graphs for the input graph with respect to all sets $U \subseteq V$ of nodes with $|U| \leq s - 1$. For each U, we can compute the block-cutpoint graph and annotate its edges in time $O(nd^2 s)$. Thus the total time for the precomputation is $O(n^s \cdot d^2 s)$, which amortizes to $O(1)$ per rule.

6 Evaluation

We evaluate the performance of our algorithms on the “Little Prince” AMR-Bank version 1.4, available from `amr.isi.edu`. This graph-bank consists of 1562 sentences manually annotated with AMRs. We implemented our algorithms in Java as part of the Alto parser for IRTGs (Alto Developers, 2015), and compared them to the Bolinas HRG parser (Andreas et al., 2013). We measured runtimes using Java 8 (for Alto) and Pypy 2.5.0 (for Bolinas) on an Intel Xeon E7-8857 CPU at 3 GHz, after warming up the JIT compilers.

As there are no freely available grammars for this dataset, we created our own for the evaluation, using Bayesian grammar induction roughly along the lines of Cohn et al. (2010). We provide the grammars as supplementary material. Around 64% of the AMRs in the graph-bank have treewidth 1 and can thus be parsed using $s = 2$ source names. 98% have treewidth 1 or 2, corresponding to $s = 3$ source names. All experiments evaluated parser times on the same AMRs from which the grammar was sampled.

Top-down versus bottom-up. Fig. 5 compares the performance of the top-down and the bottom-up algorithm, on a grammar with three source names sampled from all 1261 graphs with up to 10 nodes. Each point in the figure is the geometric mean of runtimes for all graphs with a given number of nodes; note the log-scale. We aborted the top-down parser after its runtimes grew too large.

We observe that the bottom-up algorithm outperforms the top-down algorithm, and yields practical runtimes even for nontrivial graphs. One possible explanation for the difference is that the top-down algorithm spends more time analyzing ungrammatical s-graphs, particularly subgraphs that are not connected.

Comparison to Bolinas. We also compare our implementations to Bolinas. Because Bolinas is much slower than Alto, we restrict ourselves to two source names (= treewidth 1) and sampled the grammar from 30 randomly chosen AMRs each of size 2 to 8, plus the 21 AMRs of size one.

Fig. 6 shows the runtimes. Our parsers are generally much faster than in Fig. 5, due to the decreased number of sources and grammar size. They are also both much faster than Bolinas. Measuring the total time for parsing all 231 AMRs, our bottom-up algorithm outperforms Bolinas by a factor of 6722. The top-down algorithm is slower, but still outperforms Bolinas by a factor of 340.

Further analysis. In practice, memory use can be a serious issue. For example, the decomposition grammar for s=3 for AMR #194 in the corpus has over 300 million rules. However, many uses of decomposition grammars, such as sampling for grammar induction, can be phrased purely in terms of top-down queries. The top-down algorithm can answer these without computing the entire grammar, alleviating the memory problem.

Finally, we analyzed the asymptotic runtimes in Table 1 in terms of the maximum number $d \cdot s$ of in-boundary edges. However, the top-down parser does not manipulate individual edges, but entire s-components. The maximum number D_s of s-components into which a set of s sources can split a graph is called the *s-separability* of G by Lautemann (1990). We can analyze the runtime of the top-down parser more carefully as $O(n^s 3^{D_s} ds)$; as the dotted line in Fig. 5 shows, this predicts the runtime well. Interestingly, D_s is much lower in practice than its theoretical maximum. In the

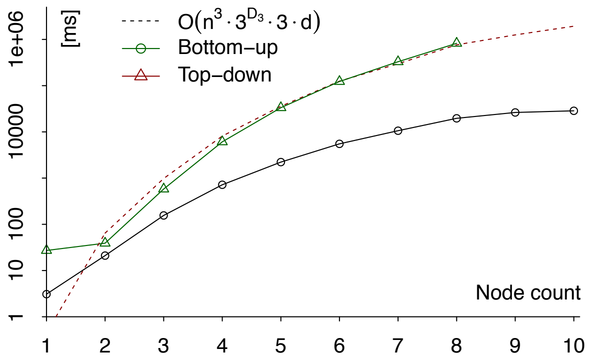

Figure 5: Runtimes of our parsers with $s = 3$.

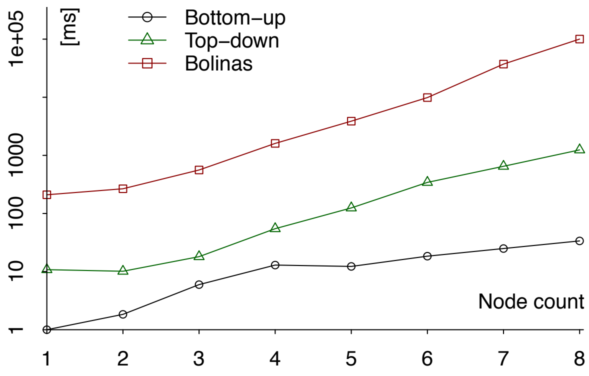

Figure 6: Runtimes of our parsers and Bolinas with $s = 2$.

"Little Prince" AMR-Bank, the mean of D_3 is 6.0, whereas the mean of $3 \cdot d$ is 12.7. Thus exploiting the s-component structure of the graph can improve parsing times.

7 Conclusion

We presented two new graph parsing algorithms for s-graph grammars. These were framed in terms of top-down and bottom-up queries to a decomposition grammar for the HR algebra. Our implementations outperform Bolinas, the previously best system, by several orders of magnitude. We have made them available as part of the Alto parser.

A challenge for grammar-based semantic parsing is grammar induction from data. We will explore this problem in future work. Furthermore, we will investigate methods for speeding up graph parsing further, e.g. with different heuristics.

Acknowledgments. We thank the anonymous reviewers for their comments, and Daniel Bauer for his help with Bolinas. We received valuable feedback at the 2015 Dagstuhl seminar on graph grammars and the 2014 Johns Hopkins workshop in Prague. This work was supported by the DFG grant KO 2916/2-1.

References

Alto Developers. 2015. Alto: algebraic language toolkit for parsing and decoding with IRTGs. Available at `https://bitbucket.org/tclup/alto`.

Jacob Andreas, Daniel Bauer, Karl Moritz Hermann, Bevan Jones, Kevin Knight, and David Chiang. 2013. Bolinas graph processing package. Available at `http://www.isi.edu/publications/licensed-sw/bolinas/`. Downloaded in January 2015.

Laura Banarescu, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer, and Nathan Schneider. 2013. Abstract meaning representation for sembanking. In *Proceedings of the 7th Linguistic Annotation Workshop & Interoperability with Discourse*, pages 178–186.

David Chiang, Jacob Andreas, Daniel Bauer, Karl Moritz Hermann, Bevan Jones, and Kevin Knight. 2013. Parsing graphs with hyperedge replacement grammars. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics*, pages 924–932.

Trevor Cohn, Phil Blunsom, and Sharon Goldwater. 2010. Inducing tree-substitution grammars. *Journal of Machine Learning Research (JMLR)*, 11:3053–3096.

Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sophie Tison, and Marc Tommasi. 2008. Tree automata techniques and applications. `http://tata.gforge.inria.fr/`.

Bruno Courcelle and Joost Engelfriet. 2012. *Graph Structure and Monadic Second-Order Logic*, volume 138 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press.

Frank Drewes, Hans-Jörg Kreowski, and Annegret Habel. 1997. Hyperedge replacement graph grammars. In Grzegorz Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation*, pages 95–162. World Scientific Publishing Co., Inc.

Jeffrey Flanigan, Sam Thomson, Jamie Carbonell, Chris Dyer, and Noah A. Smith. 2014. A discriminative graph-based parser for the abstract meaning representation. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics*, pages 1426–1436, Baltimore, Maryland.

Bevan K. Jones, Jacob Andreas, Daniel Bauer, Karl Moritz Hermann, and Kevin Knight. 2012. Semantics – Based machine translation with hyperedge replacement grammars. In *Proceedings of COLING 2012: Technical Papers*, pages 1359–1376.

Alexander Koller and Marco Kuhlmann. 2011. A generalized view on parsing and translation. In *Proceedings of the 12th International Conference on Parsing Technologies*, pages 2–13.

Alexander Koller. 2015. Semantics construction with graph grammars. In *Proceedings of the 11th International Conference on Computational Semantics (IWCS)*, pages 228–238.

Clemens Lautemann. 1988. Decomposition trees: Structured graph representation and efficient algorithms. In Max Dauchet and Maurice Nivat, editors, *13th Colloquium on Trees in Algebra and Programming*, volume 299 of *Lecture Notes in Computer Science*, pages 28–39. Springer Berlin Heidelberg.

Clemens Lautemann. 1990. The complexity of graph languages generated by hyperedge replacement. *Acta Informatica*, 27:399–421.

André F. T. Martins and Mariana S. C. Almeida. 2014. Priberam: A turbo semantic parser with second order features. In *Proceedings of the 8th International Workshop on Semantic Evaluation (SemEval 2014)*, pages 471–476.

Stephan Oepen, Marco Kuhlmann, Yusuke Miyao, Daniel Zeman, Dan Flickinger, Jan Hajic, Angelina Ivanova, and Yi Zhang. 2014. SemEval 2014 Task 8: Broad-coverage semantic dependency parsing. In *Proceedings of the 8th International Workshop on Semantic Evaluation (SemEval 2014)*, pages 63–72.

Daniel Quernheim and Kevin Knight. 2012. DAGGER: A toolkit for automata on directed acyclic graphs. In *Proceedings of the 10th International Workshop on Finite State Methods and Natural Language Processing*, pages 40–44.

Stuart M. Shieber, Yves Schabes, and Fernando C. N. Pereira. 1995. Principles and implementation of deductive parsing. *Journal of Logic Programming*, 24(1–2):3–36.