Modeling Graph Languages with Grammars Extracted via Tree Decompositions

Bevan Keeley Jones[*,†] B.K.Jones@sms.ed.ac.uk
Sharon Goldwater[*] sgwater@inf.ed.ac.uk
Mark Johnson[†] mark.johnson@mq.edu.au

∗ School of Informatics
University of Edinburgh
Edinburgh, UK

† Department of Computing
Macquarie University
Sydney, Australia

Abstract

Work on probabilistic models of natural language tends to focus on strings and trees, but there is increasing interest in more general graph-shaped structures since they seem to be better suited for representing natural language semantics, ontologies, or other varieties of knowledge structures. However, while there are relatively simple approaches to defining generative models over strings and trees, it has proven more challenging for more general graphs. This paper describes a natural generalization of the n-gram to graphs, making use of Hyperedge Replacement Grammars to define generative models of graph languages.

1 Introduction

While most work in natural language processing (NLP), and especially within statistical NLP, has historically focused on strings and trees, there is increasing interest in deeper graph-based analyses which could facilitate natural language understanding and generation applications. Graphs have a long tradition within knowledge representation (Sowa, 1976), natural language semantics (Titov et al., 2009; Martin and White, 2011; Le and Zuidema, 2012), and in models of deep syntax (Oepen et al., 2004; de Marneffe and Manning, 2008). Graphs seem particularly appropriate for representing semantic structures, since a single concept could play multiple roles within a sentence. For instance, in the semantic representation at the bottom right of Figure 1 *lake* is an argument of both *rich-in* and *own* in the sentence, "The lake is said to be rich in fish but is privately owned." However, work on graphs has been hampered, due, in part, to the absence of a general agreed upon formalism for processing and modeling such data structures. Where string and tree modeling benefits from the wildly popular Probabilistic Context Free Grammar (PCFG) and related formalisms such as Tree Substitution Grammar, Regular Tree Grammar, Hidden Markov Models, and n-grams, there is nothing of similar popularity for graphs. We need a slightly different formalism, and Hyperedge Replacement Grammar (HRG) (Drewes et al., 1997), a variety of context-free grammar for graphs, suggests itself as a reasonable choice given its close analogy with CFG. Of course, in order to make use of the formalism we need actual grammars, and this paper fills that gap by introducing a procedure for automatically extracting grammars from a corpus of graphs.

Grammars are appealing for the intuitive and systematic way they capture the compositionality of language. For instance, just as a PCFG could be used to parse "the lake" as a syntactic subject, so could a graph grammar represent *lake* as a constituent in a parse of the corresponding semantic graph. In fact, picking a formalism that is so similar to the PCFG makes it easy to adapt proven, familiar techniques for training and inference such as the inside-outside algorithm, and because HRG is context-free, parses can be represented by trees, facilitating the use of many more tools from tree automata (Knight and Graehl, 2005). Furthermore, the operational parallelism with PCFG makes it easy to integrate graph-based systems with syntactic models in synchronous grammars (Jones et al., 2012).

Probabilistic versions of deep syntactic models such as Lexical Functional Grammar and HPSG (Johnson et al., 1999; Riezler et al., 2000) are one grammar-based approach to

modeling graphs represented in the form of feature structures. However, these models are tied to a particular linguistic paradigm, and they are complex, requiring a great deal of effort to engineer and annotate the necessary grammars and corpora. It is also not obvious how to define generative probabilistic models with such grammars, limiting their utility in certain applications.

In contrast, this paper describes a method of automatically extracting graph grammars from a corpus of graphs, allowing us to easily estimate rule probabilities and define generative models. The class of grammars we extract generalize the types of regular string and tree grammars one might use to define a bigram or similar Markov model for trees. In fact, the procedure produces regular string and tree grammars as special cases when the input graphs themselves are strings or trees.

There is always overhead in learning a new formalism, so we will endeavor to provide the necessary background as simply as possible, according to the following structure. Section 2 introduces Hyperedge Replacement Grammars, which generate graphs, and their probabilistic extension, weighted HRGs. Section 3 explains how each HRG derivation of a graph induces a tree decomposition of that graph. Given a tree decomposition of a graph, we use that mapping "in reverse" to induce an HRG that generates that graph (section 4). Section 4 also introduces four different strategies for finding tree decompositions of (and hence inducing HRGs from) a set of graphs. Section 5 applies these strategies to the LOGON corpus (Oepen et al., 2004) and evaluates the induced weighted HRGs in terms of held-out perplexity. Section 6 concludes the paper and discusses possible applications and extensions.

2 Graphs and Hyperedge Replacement Grammars

Hyperedge Replacement Grammar (HRG) is a generalization of CFG to graph languages (see Drewes et al. (1997) for an overview). Where a CFG builds up strings by replacing symbols with new substrings, an HRG builds graphs by replacing edges with subgraphs. As a context-free formalism, HRG derivations can be described by trees, similar to CFG parses. Thus, in the case of probabilistic HRG, it is possible to assign rule weights to define easily factorizable probability distributions over graphs, just as PCFGs do for strings.

We start by defining a hypergraph, a generalization of a graph where edges may link any finite number of vertices. Formally, a hypergraph is a tuple $(\mathcal{V}, \mathcal{E}, \alpha, \ell, x)$. $\mathcal{V}$ and $\mathcal{E}$ are finite sets of vertices and hyperedges, respectively. The *attachment function* $\alpha : \mathcal{E} \to \mathcal{V}^*$ maps each hyperedge $e \in \mathcal{E}$ to a sequence of pairwise distinct vertices from $\mathcal{V}$, where we call the length of $\alpha(e)$ the *arity* of e. The *labeling function* $\ell : \mathcal{E} \to \Sigma$ maps each hyperedge to a symbol in some ranked alphabet Σ, where the rank of $\ell(e)$ is e's arity. Vertices are unlabeled, but they can be simulated by treating unary hyperedges (i.e., hyperedges with a single vertex) as vertex labels. Finally, each graph has a set of zero or more *external vertices*, arranged in a sequence $x \in \mathcal{V}^*$ (pairwise distinct), which plays an important role in the rewriting mechanism of HRG. Just as hyperedges have an arity, so too do hypergraphs, defined as the length of x.

We are primarily interested in languages of simple directed graphs, hypergraphs where each edge is either binary or, for vertex labels, unary. In this case, we can indicate visually the ordering on a binary edge with vertex sequence $v_0 v_1$ by an arrow pointing from vertex v_0 to v_1. We may make use of hyperedges of arbitrary arity, though, for intermediate rewriting steps during derivations. The semantic dependency graph at the bottom right of Figure 1, taken from the Redwoods corpus (Oepen et al., 2004), is an example of a simple graph. It has both unary edges for expressing predicates like 'private' and 'own' and binary edges for specifying their relations. In principle, any vertex can have more than one unary edge, a fact we make use of in HRG rule definitions, such as in the graph on the right-hand side of rule $r4$ in Figure 1 where vertex 2 has two unary edges labeled 'rich-in' and $N_{\text{rich-in}}$.

A weighted HRG is an edge rewriting system for generating hypergraphs, also defined as a tuple $(\Sigma, \mathcal{N}, S, \mathcal{R})$. Σ is a ranked alphabet of edge labels, $\mathcal{N} \subset \Sigma$ a set of nonterminal symbols, $S \in \mathcal{N}$ a special start symbol, and $\mathcal{R}$ is a finite set of weighted rules. Each rule

in $\mathcal{R}$ is of the form $[A \rightarrow h].w$, where h is a hypergraph with edge labels from Σ, $A \in \mathcal{N}$ has rank equal to the arity of h, and weight w is a real number. As with PCFGs, a weighted HRG is probabilistic if the weights of all rules with the same ranked symbol A on the left-hand side sum to one. In the case of probabilistic HRG, the probability of a derivation is the product of the weights of the rules in the derivation, just as for PCFG. Figure 1 shows an example of an HRG and a sample derivation. The external vertices of the right-hand side graphs have been shaded, and their sequence should be read top to bottom (e.g., 0 to 5 in rule $r1$). Vertices have been identified by numbers, but these identifiers are included only to make it easier to refer to them in our discussion; strictly speaking, vertices are unlabeled, and these numbers are irrelevant to the operation of the grammar. Nonterminal edges are dashed to make them easier to identify.

Hyperedge replacement, the basic rewriting mechanism of HRG, is an operation where a hypergraph is substituted for an edge. If g is a hypergraph containing edge e, and h is another hypergraph with the same arity as e, edge e can be replaced with h by first removing e from g and then "fusing" h and g together at the external vertices of h and the vertices of $\alpha(e)$. So, if $\alpha(e) = v_0 v_1 ... v_k$ and h has external vertices $u_0 u_1 ... u_k$, we would fuse each u_i to the corresponding v_i.

Much like with CFG, where each step of a derivation replaces a symbol by a substring, each step of an HRG derivation replaces an edge with a certain nonterminal symbol label by the right-hand side graph of some rule with the same symbol on its left-hand side. For instance, in the application of rule $r3$ in the fourth step of Figure 1, the edge $1 \overset{N_{\text{say}}}{\rightarrow} 5$ is replaced by the graph $1 \overset{\text{arg2}}{\rightarrow} 2 \overset{N_{\text{arg2}}}{\rightarrow} 5$ by removing the red N_{say} edge and then attaching the new subgraph. Rule $r3$ has an external vertex sequence of 1 to 5, and these are fused to the incident vertices of the nonterminal edge $1 \overset{N_{\text{say}}}{\rightarrow} 5$. The edge to be replaced in each step has been highlighted in red to ease reading.

3 Tree Decompositions

We now introduce one additional piece of theoretical machinery, the *tree decomposition* (Bodlaender, 1993). Tree decompositions play an important role in graph theory, feature prominently in the junction tree algorithm from machine learning (Pearl, 1988), and have proven valuable for efficient parsing (Gildea, 2011; Chiang et al., 2013). Importantly, Lautemann (1988) proved that every HRG parse identifies a particular tree decomposition, and by restricting ourselves to a certain type of tree we will draw an even tighter relationship, allowing us to identify parses given tree decompositions.

A tree decomposition of a graph g is a tree whose nodes identify subsets of the vertices of g which satisfy the following three properties:[1]

- **Vertex Cover**: Every vertex of g is contained by at least one tree node.
- **Edge Cover**: For every edge e of the graph, there is a tree node η such that each vertex of $\alpha(e)$ is in η.
- **Running Intersection**: Given any two tree nodes η_0 and η_1, both containing vertex v, all tree nodes on the unique path from η_0 to η_1 also contain v.

Figure 2 presents four different tree decompositions of the graph shown at the bottom right of Figure 1. Consider (d). Vertex cover is satisfied by the fact that every vertex of the graph appears in at least one tree node. Graph vertex 0, for example, is covered by two nodes $\{0, 1, 5\}$ and $\{0, 3, 5\}$. Similarly, every edge is covered by at least one of the nodes. Node $\{0, 1, 5\}$ covers one binary edge, $0 \overset{\text{arg1}}{\rightarrow} 1$, and three unary edges: $\overset{\text{but}}{0}$, $\overset{\text{say}}{1}$, and $\overset{\text{lake}}{5}$.

We focus on a particular class called *edge-mapped* tree decompositions, defined by pairs (t, μ) where t is a tree decomposition of some graph g and μ is a bijection from the nodes of t to the edges of g, where a node also covers the edge it maps to. Every graph has at least one edge-mapped tree decomposition; Figure 2(a)-(c) illustrates three such edge-mapped decompositions for a particular graph, where the mapping is shown by the extra labels next to the tree nodes. The edge mapping simplifies the rule extraction procedure described in Section 4 since traversing the tree and following

[1] To avoid confusion, we adopt a terminology where *node* is always used in respect to tree decompositions and *vertex* and *edge* to graphs.

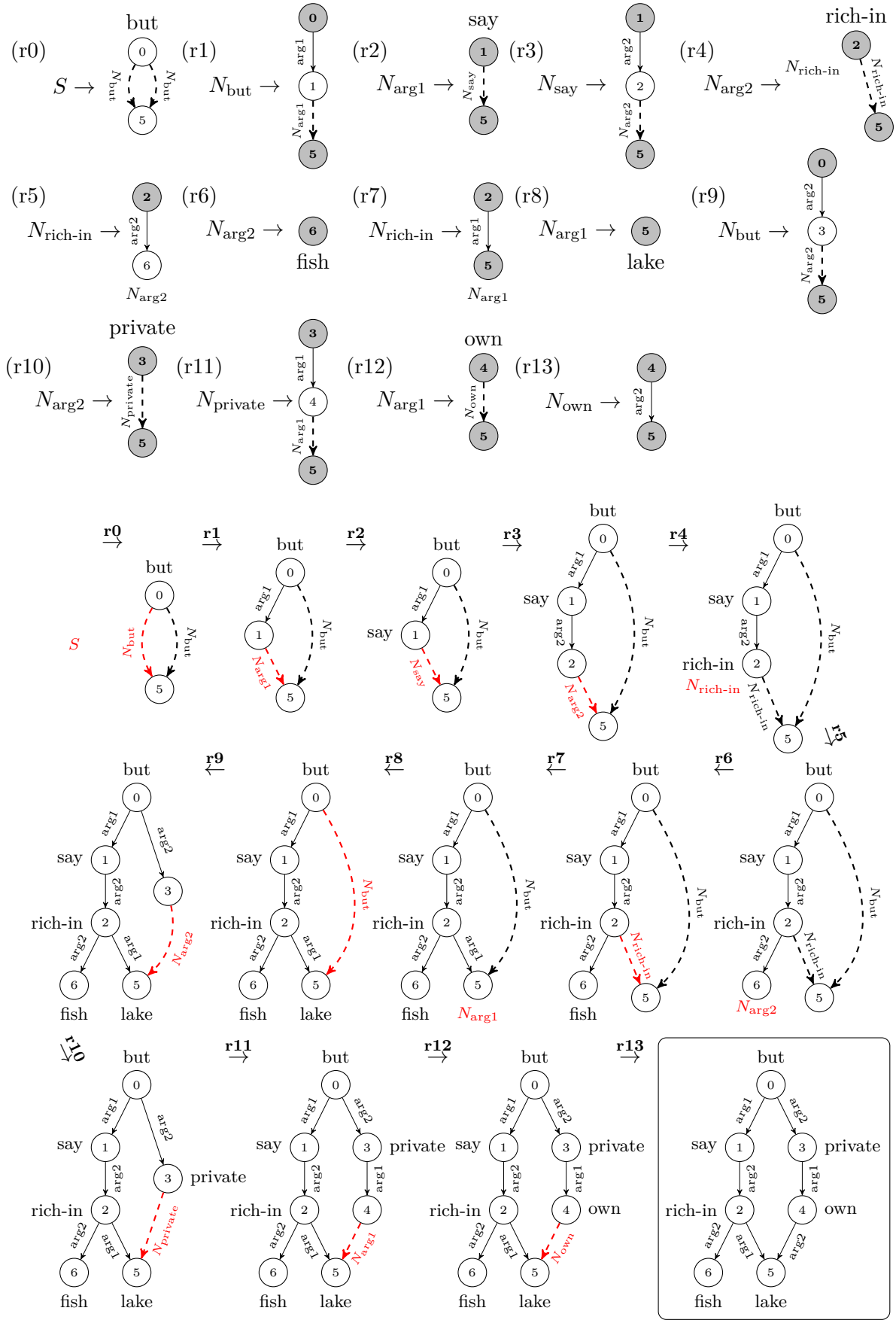

Figure 1: An HRG and a derivation of the semantic dependency graph for "the lake is said to be rich in fish but is privately owned." External vertices are shaded and ordered top to bottom, nonterminal edges are dashed, and the one being replaced is highlighted in red.

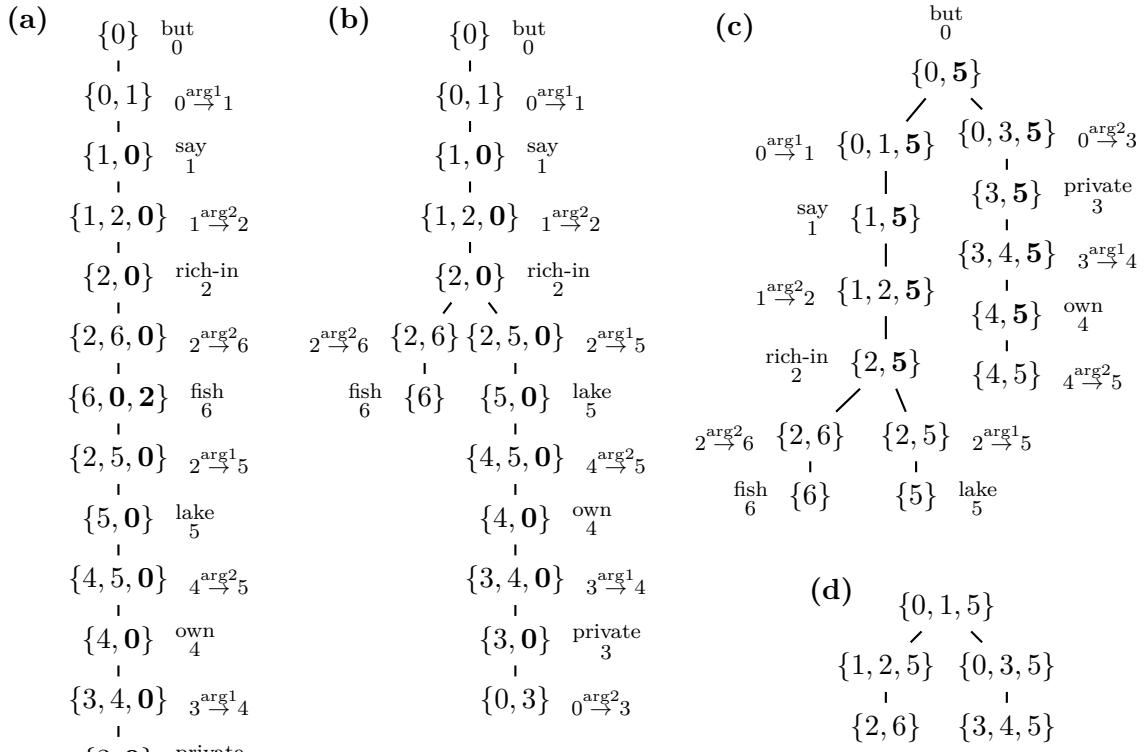

Figure 2: Edge-mapped (a-c) and non-edge-mapped (d) tree decompositions for the graph at the bottom right of Figure 1.

this mapping μ guarantees that every graph edge is visited exactly once.

Running intersection will also prove important for rule extraction, since it tracks the tree violations of the graph by passing down the end points of edges that link edges in different branches of the decomposition. This same information must be passed down the respective paths of the HRG derivation tree via the external vertices of rule-right hand sides. Figure 2 uses bold face and vertex order to highlight the vertices that must be added to each node beyond those needed to cover its corresponding edge. In the decomposition shown in (b), vertex 0 must be passed from the node mapping to $\overset{\text{but}}{0}$ down to the node mapping to $0 \overset{\text{arg2}}{\rightarrow} 3$ because the two edges share that vertex. Any HRG derivation will need to pass down vertices in a similar manner to specify which edges get attached to which vertices.

As suggested by the four trees of Figure 2, there are always many possible decompositions for any given graph. In the next section we describe three methods of producing tree decompositions, each leading to distinct grammars with different language modeling properties.

Algorithm 1: Extract HRG rule $A \rightarrow h$ from tree decomposition node η.

function EXTRACT(η)
  $A \leftarrow \text{label}(\text{parent}(\eta), |\text{parent}(\eta) \cap \eta|)$
  $h.x \leftarrow \text{order}(\eta, \text{parent}(\eta) \cap \eta)$
  add terminal edge $\mu(\eta)$ to h
  for all $\eta_i \in \text{children}(\eta)$ **do**
    add nonterminal edge u_i to h
    $\alpha(u_i) \leftarrow \text{order}(\eta_i, \eta \cap \eta_i)$
    $\ell(u_i) \leftarrow \text{label}(\eta, |\eta \cap \eta_i|)$
  return $[A \rightarrow h]$

4 HRG Extraction

Rule extraction proceeds by first selecting a particular tree decomposition for a graph and then walking this tree to extract grammar rules in much the same way as one extracts n-grams or Regular Tree Grammars (RTG) from a corpus of strings or trees. The procedure (Algorithm 1) extracts a single rule for each node of the decomposition to generate the associated terminal edge plus a set of nonterminals which can be subsequently expanded to

generate the subgraphs corresponding to each subtree of the decomposition node. In particular, given the tree decomposition in Figure 2(c), the procedure produces the grammar in Figure 1. Rule extraction works for any connected simple graph and can be easily adapted for arbitrary hypergraphs.

Start by assigning the left-hand side nonterminal symbol according to

$$\text{label}(\text{parent}(\eta), r),$$

which returns a symbol determined by η's parent with rank r, the number of vertices in common between η and its parent. The external vertices of h are assigned by sorting the vertices that η shares with its parent. Any ordering policy will work so long as it produces the same ordering with respect to a given decomposition node. What is important is that the order of the external vertices of a rule match that of the vertices of the nonterminal edge it expands.[2] The algorithm then constructs the rest of h by including terminal edge $\mu(\eta)$, and adding a nonterminal edge for each child η_i of η, with vertices assigned according to an ordering of the vertices that η shares with η_i, again labeled according to *label*.

The function *label* just returns a nonterminal symbol of a given rank, chosen to match the number of external vertices of the right-hand side. There are many possible choices of *label*; it can even be a function that always returns the same symbol for a given rank. For purposes of language modeling, it is useful to condition rule probabilities on the label of the edge associated with the parent node in the decomposition (analogous to conditioning on the preceding word in a bigram setting). It is also useful to distinguish the direction of that preceding edge. For instance, we would expect 'rich-in' to have a different probability based on whether it is being generated as the argument of predicate 'say' vs. as a descendant of its own argument 'lake'. Thus, each nonterminal encodes (1) the label of the preceding edge and (2) its direction with respect to the current edge as defined according to the *head-to-tail* relation, where we say edge e_j is head-to-tail with preceding edge e_i iff the last vertex of $\alpha(e_i)$ is the first of $\alpha(e_j)$. For instance, $\overset{\text{lake}}{5}$ is in head-to-tail relation with $2 \overset{\text{arg1}}{\rightarrow} 5$, while $2 \overset{\text{arg1}}{\rightarrow} 5$ is not head-to-tail with $4 \overset{\text{arg2}}{\rightarrow} 5$.

The grammar in Figure 1 is extracted according to the tree decomposition in Figure 2(c). Consider how rule $r4$ is constructed while visiting the node $\eta = \{2, 5\}$ which maps to unary edge $\overset{\text{rich-in}}{2}$. The left-hand side symbol N_{arg2} comes from the label of the edge $1 \overset{\text{arg2}}{\rightarrow} 2$ associated with η's parent node $\{1, 2, 5\}$ and has a rank of $|\{2, 5\} \cap \{1, 2, 5\}| = 2$. The rule right-hand side is constructed so that it contains $\overset{\text{rich-in}}{2}$ and two nonterminal edges. The first nonterminal edge comes from the intersection of η with left child $\{2, 6\}$, yielding unary sequence 2 and edge $\overset{N_{\text{rich-in}}}{2}$. The second nonterminal edge is constructed similarly by ordering the vertices in the intersection of η with its right child $\{2, 5\}$ to get the binary sequence 2 to 5, producing $2 \overset{N_{\text{rich-in}}}{\rightarrow} 5$. Finally, the external vertex sequence comes from ordering the members of $\{2, 5\} \cap \{1, 2, 5\}$.

The particular edge-mapped tree decomposition plays a key role in the form of the extracted rules. In particular, each branching of the tree specifies the number of nonterminals in the corresponding rule. For example, decompositions such as Figure 2(a) result in linear grammars, where every rule right-hand side contains at most one nonterminal.

We experiment with three different strategies for producing edge-mapped tree decompositions. In each case, we start by building a node-to-edge map by introducing a new tree node to cover each edge of the graph, simultaneously ensuring the vertex and edge cover properties. The strategies differ in how the nodes are arranged into a tree. One simple approach (linear) is to construct a linearized sequence of edges by performing a depth first search of the graph and adding edges when we visit incident vertices. This produces non-branching trees such as Figure 2(a). Alternatively, we can construct the decomposition according to the actual depth first search *tree* (dfs), producing decompositions like (b). Finally, we construct what we call a topological sort tree (top), where we add children to each node so as to maximize the number of head-

[2] We experimented with various orderings, from pre-order traversals of the tree decomposition to simply sorting by vertex identity, all with similar results.

to-tail transitions, producing trees such as (c). For rooted DAGs, this is easy; just construct a directed breadth first search tree of the graph starting from the root vertex. It is more involved for other graphs but still straightforward, accomplished by finding a minimum spanning tree of a newly constructed weighted directed graph representing head-to-tail transitions as arcs with weight 0 and all other contiguous transitions as arcs of weight 1. Once the edge-mapped nodes are arranged in a tree all that is left is to add vertices to each to satisfy running intersection.

One attractive feature of *top* is that, for certain types of input graphs, it produces grammars of well-known classes. In particular, if the graph is a string (a directed path), the grammar will be a right-linear CFG, i.e., a regular string grammar (a bigram grammar, in fact), and if it is a rooted tree, the unique topological sort tree leads to a grammar that closely resembles an RTG (where trees are edge-labeled and siblings are un-ordered). The other decomposition strategies do not constrain the tree as much, and their grammars are not necessarily regular.

Another nice feature of *top* is that subtrees of a parse tend to correspond to intuitive modules of the graph. For instance, the grammar first generates a predicate like 'rich-in' and then it proceeds to generate the subgraphs corresponding to its arguments 'fish' and 'lake', much as one would expect a syntactic dependency grammar to generate a head followed by its dependents. The linear grammar derived from Figure 2(a), on the other hand, would generate 'lake' as a descendant of 'fish'.

We also explore an augmentation of *top* called the *rooted* topological sort tree (r-top). Any graph can be converted to a rooted graph by simply adding an extra vertex and making it the parent of every vertex of in-degree zero (or if there are none, picking a member of each connected component at random). We exploit this fact to produce a version of *top* that generates all graphs as though they were rooted by starting off each derivation with a rule that generates every vertex with in-degree zero. We expect rooted graphs to produce simpler grammars in general because they reduce the number of edges that must be generated in non-topological order, requiring fewer rules that differ primarily in whether they generate an edge in head-to-tail order or not. In particular, if a graph is acyclic, all edges will be generated in head-to-tail relation and the corresponding grammar will contain fewer nonterminals.

5 Evaluation

We experiment using 5564 elementary semantic dependency graphs taken from the LOGON portion of the Redwoods corpus (Oepen et al., 2004). From Table 1, we can see that, while there are a few tree-shaped graphs, the majority are more general DAGs. Nevertheless, edge density is low; the average graph contains about 15.4 binary edges and 14.9 vertices. We set aside every 10^{th} graph for the test set, and estimate the models from the remaining 5,008, replacing terminals occurring ≤ 1 times in the training set with special symbol *UNK*.

Model parameters are calculated from the frequency of extracted rules using a mean-field Variational Bayesian approximation of a symmetric Dirichlet prior with parameter β (Bishop, 2006). This amounts to counting the number of times each rule r with left-hand side symbol A is extracted and then computing its weight θ_r according to

$$\theta_r = \exp\left(\Psi(n_r + \beta) - \Psi(\sum_{r':r'=A\rightarrow h} n_{r'} + \beta)\right),$$

where n_r is the frequency of r and Ψ is the standard digamma function. This approximation of a Dirichlet prior offers a simple yet principled way of simultaneously smoothing rule weights and incorporating a soft assumption of sparsity (i.e., only a few rules should receive very high probability). Specifically, we somewhat arbitrarily selected a value of 0.2 for β, which should result in a moderately sparse distribution.

We evaluate each model by computing perplexity: $2^{-\sum_{i=1}^{N} \frac{1}{N} \ln_2 p(g_i)}$, where N is the number of graphs in the test set, g_i is the i^{th} graph, and $p(g_i)$ is its probability according to the model, computed as the product of the weights of the rules in the extracted derivation. Better models should assign higher probability to g_i, thereby achieving lower perplexity.

(a) Graphs

strings	0
r-trees	682
r-dags	51
dags	4831
total	**5564**

(b) Edges

	unary	binary
types	5626	10
tokens	83061	85737

Table 1: LOGON corpus. (a) Graph types (r stands for rooted). (b) Edge types and tokens.

model	perplexity	size
linear	505,061	59,980
dfs	341,336	14,443
top	22,484	20,985
r-top	40,504	19,052

Table 2: Model perplexity and grammar size.

Table 2 lists the perplexities of the language models defined according to our four different tree decomposition strategies. *Linear* is relatively poor since it makes little distinction between local and more distant relations between edges. For instance, the tree in Figure 2(a) results in a grammar where $2 \overset{\text{arg2}}{\rightarrow} 5$ is generated as the child of distantly related $\overset{\text{fish}}{6}$ but as a remote descendant of neighboring edge $\overset{\text{rich-in}}{2}$. *Dfs* is better, but suffers from similar problems. Both *top* and *r-top* perform markedly better, but *r-top* less so because the initial rule required for generating all vertices of indegree zero is often very improbable. There are 1562 different such rules required for describing the training data, many of which appear only once. We believe there are ways of factorizing these rules to mitigate this sparsity effect, but this is left to future work.

Grammar sizes are also somewhat telling. The linear grammar is quite large, due to the extra rules required for handling the long-distance relations. The other grammars are of a similar, much smaller size, but *dfs* is smallest since it tends to produce trees of much smaller branching factor, allowing for greater rule reuse. As predicted, the *r-top* grammar is somewhat smaller than the vanilla *top* grammar, but, as previously noted, the potential reduction in sparsity is counteracted by the introduction of the extra initial rules.

6 Conclusion & Discussion

Graph grammars are an appealing formalism for modeling the kinds of structures required for representing natural language semantics, but there is little work in actually defining grammars for doing so. We have introduced a simple framework for automatically extracting HRGs, based upon first defining a tree decomposition and then walking this tree to extract rules in a manner very similar to how one extracts RTG rules from a corpus of trees. By varying the kinds of tree decomposition used, the procedure produces different types of grammars. While restricting consideration to a broad class of tree decompositions where visiting tree nodes corresponds to visiting edges of the graph, we explored four special cases, demonstrating that one case, where parent-to-child node relations in the tree maximize head-to-tail transitions between graph edges, performs best in terms of perplexity on a corpus of semantic graphs. This topological ordering heuristic seems reasonable for the corpus we experimented on since such parent-child transitions are equivalent to predicate-argument transitions in the semantic representations.

Interesting questions remain as to which particular combinations of graph and decomposition types lead to useful classes of graph grammars. In our case we found that our topological sort tree decomposition leads to regular grammars when the graphs describe strings or particular kinds of trees, making them useful for defining simple Markov models and also making it possible to perform other operations like language intersection (Gecseg and Steinby, 1984). We have presented only an initial study and there are potentially many interesting combinations.

Acknowledgments

We thank Mark-Jan Nederhof for his comments on an early draft and the anonymous reviewers for their helpful feedback. This research was supported in part by a prize studentship from the Scottish Informatics and Computer Science Alliance, the Australian Research Council's Discovery Projects funding scheme (project numbers DP110102506 and DP110102593) and the US Defense Advanced Research Projects Agency under contract FA8650-11-1-7151.

References

Christopher M. Bishop. 2006. *Pattern Recognition and Machine Learning*. Springer.

Hans L. Bodlaender. 1993. A tourist guide through treewidth. *Acta Cybernetica*, 11(1-2):1–21.

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 Meeting of the ACL*.

Marie-Catherine de Marneffe and Christopher D. Manning. 2008. The stanford typed dependencies representation. In *Proceedings of COLING Workshop on Cross-framework and Cross-domain Parser Evaluation*.

Frank Drewes, Annegret Habel, and Hans-Jorg Kreowski. 1997. Hyperedge replacement graph grammars. *Handbook of Graph Grammars and Computing by Graph Transformation*, pages 95–1626.

Ferenc Gecseg and Magnus Steinby. 1984. *Tree Automata*. Akademiai Kiado, Budapest.

Daniel Gildea. 2011. Grammar factorization by tree decomposition. *Computational Linguistics*, 37(1):231–248.

Mark Johnson, Stuart Geman, Stephen Canon, Zhiyi Chi, and Stefan Riezler. 1999. Estimators for stochastic 'unification-based' grammars. In *Proceedings of the 37th Meeting of the ACL*, pages 535–541.

Bevan Jones, Jacob Andreas, Daniel Bauer, Karl-Moritz Hermann, and Kevin Knight. 2012. Semantics-based machine translation with hyperedge replacement grammars. In *Proceedings of COLING*.

Kevin Knight and Jonathon Graehl. 2005. An overview of probabilistic tree transducers for natural language processing. In *Proceedings of the 6th International Conference on Intelligent Text Processing and Computational Linguistics*.

Clemens Lautemann. 1988. Decomposition trees: Structured graph representation and efficient algorithms. In M. Dauchet and M. Nivat, editors, *CAAP '88*, volume 299 of *Lecture Notes in Computer Science*, pages 28–39. Springer Berlin Heidelberg.

Phong Le and Willem Zuidema. 2012. Learning compositional semantics for open domain semantic parsing. In *Proceedings of COLING*.

Scott Martin and Michael White. 2011. Creating disjunctive logical forms from aligned sentences for grammar-based paraphrase generation. In *Proceedings of the Workshop on Monolingual Text-To-Text Generation (MTTG)*, pages 74–83.

Stephan Oepen, Dan Flickinger, Kristina Toutanova, and Christopher D. Manning. 2004. Lingo redwoods. *Research on Language and Computation*, 2(4):575–596.

Judea Pearl. 1988. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, San Francisco, CA, 2 edition.

Stefan Riezler, Detlef Prescher, Jonas Kuhn, and Mark Johnson. 2000. Lexicalized stochastic modeling of constraint-based grammars using log-linear measures and em. In *Proceedings of the 38th Meeting of the ACL*.

John F. Sowa. 1976. Conceptual graphs for a data base interface. *IBM Journal of Research and Development*, 20(4):336–357.

Ivan Titov, James Henderson, Paola Merlo, and Gabriele Musillo. 2009. Online graph planarisation for synchronous parsing of semantic and syntactic dependencies. In *Proceedings of IJCAI*, pages 1562–1567.