Ordered Tree Decomposition for HRG Rule Extraction

Daniel Gildea
University of Rochester
Computer Science Department
gildea@cs.rochester.edu

Giorgio Satta
Università di Padova
Dipartimento di Ingegneria
dell'Informazione
satta@dei.unipd.it

Xiaochang Peng
University of Rochester
Computer Science Department
xpeng@cs.rochester.edu

We present algorithms for extracting Hyperedge Replacement Grammar (HRG) rules from a graph along with a vertex order. Our algorithms are based on finding a tree decomposition of smallest width, relative to the vertex order, and then extracting one rule for each node in this structure. The assumption of a fixed order for the vertices of the input graph makes it possible to solve the problem in polynomial time, in contrast to the fact that the problem of finding optimal tree decompositions for a graph is NP-hard. We also present polynomial-time algorithms for parsing based on our HRGs, where the input is a vertex sequence and the output is a graph structure. The intended application of our algorithms is grammar extraction and parsing for semantic representation of natural language. We apply our algorithms to data annotated with Abstract Meaning Representations and report on the characteristics of the resulting grammars.

1. Introduction

In statistical natural language processing, parsing has been traditionally intended as the task of taking as input a string, representing a sequence of word tokens, along with a grammar, and producing as output one or possibly more trees that hierarchically group those tokens according to the grammar's rules. In this article we use the term **string-to-tree parsing** when referring to this task, in contrast with a different parsing problem

Submission received: 18 May 2018; revised version received: 24 February 2019; accepted for publication: 4 March 2019.

doi:10.1162/COLI_a_00350

that will be introduced later, in which the produced output is a general graph. The high relevance of string-to-tree parsing in statistical natural language processing stems from a historical focus on syntactic analysis and on formal grammars that produce tree structures, as for instance context-free grammars (CFGs) or dependency grammars. This perspective has been strongly influenced by the fact that generative linguistic theories use hierarchical structures to model the syntax of natural language. In the last decades, string-to-tree parsing has also been reinforced by the proliferation of syntactically annotated data for several languages, available in the form of treebanks.

At the time of writing there is growing attention toward semantic analysis of sentences with the conviction that, along with syntactic information, semantic information will greatly improve end-user applications. Whereas, in statistical natural language processing, semantic analysis had been mainly investigated by means of separated tasks (such as named entity recognition, semantic role labeling, and co-reference resolution), we now witness the attempt to incorporate all these different aspects into a single process of semantic parsing. This trend is attested to by the wide variety of semantic representations that are being proposed (Kuhlmann and Oepen 2016) and by the number of semantically annotated data sets being produced and distributed. As a follow-up, semantic parsing is now being exploited in machine translation (Jones et al. 2012), text summarization (Liu et al. 2015; Dohare and Karnick 2017), sentence compression (Takase et al. 2016), event extraction (Huang et al. 2016; Rao et al. 2017; Wang et al. 2017), and question answering (Khashabi et al. 2018). It seems therefore that the field is returning to the task of semantic parsing after a long hiatus, since semantic analysis was already part of the agenda in formalisms such as head-driven phrase structure grammars in the era when grammars were generally manually developed rather than learned from data.

Crucially, semantic representations are graph structures, as opposed to tree structures. There is therefore a need for formal grammars that generate graphs. Graph grammars have been investigated in the formal language literature since the mid-1970s. These models are based on a derive relation that rewrites graphs, and therefore they generate graph languages, that is, languages that are sets of graphs. Accordingly, the problem of **graph parsing** has been conceived in such a way that a graph structure and a graph grammar are provided as input, and the goal is to produce as output one or more grammar derivations that generate the input graph itself. However, this problem setting does not apply to the scenario of semantic parsing for natural language. More precisely, in semantic parsing the input is a graph grammar along with a sentence represented as a sequence of tokens that, to simplify our discussion, can be regarded as the vertices of some graph. One then needs to output one or more graphs, defined over the input vertices, that can be generated by the input grammar. These graphs thus represent possible semantic representations associated with the input sentence. In this article we use the term **string-to-graph parsing** when referring to this novel problem.

We observe here that graph parsing and string-to-graph parsing, as just described, are incomparable problems. More precisely, in graph parsing the input graph is fully specified by means of its vertices and edges, whereas in string-to-graph parsing we do not have access to the edges of the graph. Therefore, in this respect, string-to-graph is more general than graph parsing. On the other hand, in string-to-graph parsing we are given a specific ordering of the graph vertices, which is not provided in graph parsing, and we might take advantage of such restriction for graph grammars of certain types. To our knowledge, there is no relevant work in the literature on graph grammars investigating string-to-graph parsing. This article provides a contribution in this direction.

String-to-tree parsing is rather well understood. If we assume that when generating a string our trees do not have crossing arcs,[1] then string-to-tree parsing can be efficiently solved using dynamic programming algorithms, also called chart-based methods or tabular methods in the natural language processing community. For an introduction to dynamic programming methods for string-to-tree parsing, we refer the reader to Graham and Harrison (1976) and Nederhof and Satta (2004) for CFGs, and to Kübler, McDonald, and Nivre (2009, Chapter 5) for projective dependency grammars. Dynamic programming is perhaps the most general way to attack string-to-tree parsing, and other alternative methods such as greedy algorithms or beam search algorithms can be derived as special cases of dynamic programming, as discussed by Huang and Sagae (2010) and Kuhlmann, Gómez-Rodríguez, and Satta (2011). In most cases of interest, dynamic programming provides polynomial space and time solutions for non-crossing string-to-tree parsing as well as for unsupervised learning of the weights of grammar rules from annotated data, assuming one is interested in data-driven methods.

Note that string-to-tree parsing and string-to-graph parsing are two closely related problems, because they both take as input a sequence of tokens and a grammar, and produce as output a structure that can be generated by the grammar and is "compatible" with the input sequence. Because of this similarity, algorithms for string-to-tree parsing that use dynamic programming can be naturally extended to string-to-graph parsing. However, when we do so, we soon realize that there is a gap in computational complexity between the two cases, because of the structural differences between non-crossing trees and general graphs. More precisely, the trees considered by the parsing algorithms mentioned here do not have crossing arcs. In contrast, the graphs associated with the semantic structures we are interested in have arcs that can cross each other, relative to the input sentence. Furthermore, if we assume that the arcs in our two structures are directed, a tree node has at most one parent, whereas a graph node might have more than one parent. Because of crossing arcs and/or multi-parenthood, extending existing string-to-tree parsing algorithms to string-to-graph parsing requires additional bookkeeping, resulting in a considerable increase in computational resources. We illustrate this problem by means of a concrete example involving dependency structures.

Example 1
We consider general dependency structures, that is, graph-like structures where each vertex represents a word in the input. We informally compare the application of dynamic programming to projective (i.e., non-crossing) string-to-tree parsing and string-to-graph parsing for such structures. Most important, our discussion abstracts away from the specific class of formal grammars used to generate our dependency structures, and focuses essentially on the two structural features of crossing arcs and multi-parenthood, which distinguish general graphs from projective trees, as already discussed.

We start with string-to-tree parsing. Let $w = a_1 a_2 \cdots a_n$ be the input string. At the heart of dynamic programming algorithms for projective, dependency tree parsing, we find a set of operations to combine two trees, resulting in a larger tree. When combining trees, the only information we care about is the span and the root of each tree. The span is needed to make sure that trees to be combined do not overlap and that the combination forms a tree spanning a larger (connected) substring of w. The root is used to compute the score of the added dependency arc, which contributes to the

1 In the natural language processing literature, trees with crossing arcs are also called discontinuous trees in the context of phrase structures, or else non-projective trees in the context of dependency structures.

score of the resulting tree. Because the span of a tree t is a substring of w of the form $a_{i+1}a_{i+2}\cdots a_j$, we may conveniently record t's span by means of the two indices i and j. Furthermore, the root of t is a word a_h occurring in $a_{i+1}a_{i+2}\cdots a_j$, and we may similarly record this information by means of the index h.

Consider now two trees t and t', and assume that they share the same span i, j and the same root h. We can conveniently group t and t' into the same equivalence class $[i, j; h]$. We call each equivalence class a **partial analysis**. Figure 1(a) graphically represents partial analysis $[i, j; h]$. In order to combine trees, we can then use partial analyses. This is exemplified in Figure 1(b), where we show the basic operation of left tree attachment. Each tree in $[i, k; h']$ is attached as a left dependent of the root of each tree in $[k, j; h]$ by means of a new arc with head h and dependent h'. Each attachment provides a new tree with span i, j and root h, and the resulting trees are represented in the new partial analysis $[i, j; h]$. A second operation of right tree attachment is symmetrically defined.

The representation of trees by means of partial analyses is at the root of the efficiency of the algorithm, which we now briefly analyze. Each index i, j, and h in a partial analysis can range from 0 to n, the length of w, for a total number of partial analyses in $\mathcal{O}(n^3)$. Furthermore, the left/right tree attachment operations can be implemented in constant time for fixed values of i, k, j, h, and h'. Then the overall time used by the algorithm when combining trees is in $\mathcal{O}(n^5)$. We conclude that our parsing algorithm runs in polynomial time and space. What we have briefly summarized here is a well-known adaptation of the standard Cocke-Kasami-Younger (CKY) algorithm for parsing using CFGs (Aho and Ullman 1972), as applied to lexicalized CFGs (Collins 1996) and to dependency grammars (Eisner 1996).

Let us now turn to string-to-graph parsing. When applying dynamic programming, we can exploit ideas similar to those presented for the string-to-tree case, but we need to make some generalizations. The input to the algorithm is now a sequence of

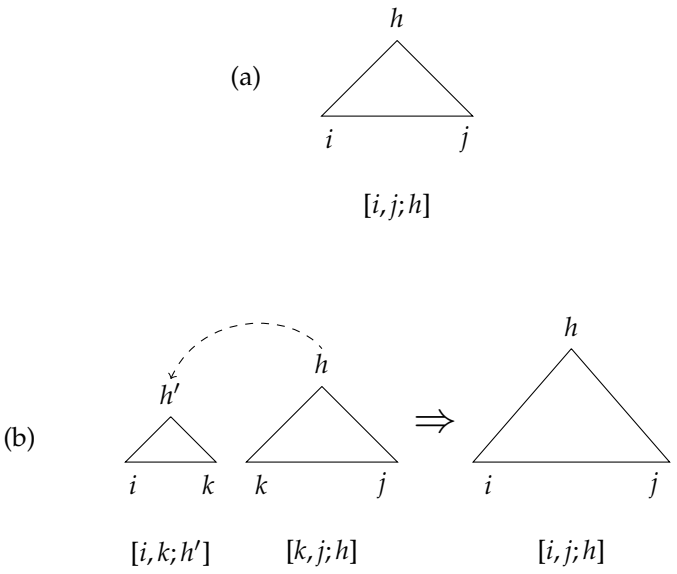


Figure 1
Graphical representation for (a) the partial analysis $[i, j; h]$ and (b) the combination of partial analyses $[i, k; h']$ and $[k, j; h]$, implementing left attachment and producing partial analysis $[i, j; h]$.

vertices, which we again represent as $w = v_1 v_2 \cdots v_n$. We need to combine graphs into larger graphs, eventually leading to a graph whose set of vertices spans all of w. In general, we need to deal with intermediate graphs G whose span may not be a single substring of w, as in the case of projective dependency trees, but rather several unconnected non-empty substrings of the form $s_1 = v_{i_1+1} v_{i_1+2} \cdots v_{i_2}$, $s_2 = v_{i_3+1} v_{i_3+2} \cdots v_{i_4}$, ..., $s_{q/2} = v_{i_{q-1}+1} v_{i_{q-1}+2} \cdots v_{i_q}$, for some even $q > 2$. This is a consequence of the fact that our graphs might have crossing arcs that reach the vertices placed in between the s_i's. We then represent G's span by means of the associated indices $i_1, i_2, \ldots, i_q$. Furthermore, whereas for dependency trees a subtree is connected to the rest of the structure only through its root, G will be connected to the final graph that we want to construct through *several* of its vertices. This is a consequence of the fact that the nodes of our graphs can have multiple parent nodes. We call the connecting vertices the **attachment vertices** of G, and again denote this set by means of the associated indices $h_1, h_2, \ldots, h_d$, for some $d \geq 1$.

When combining graphs, the span and the attachment vertices are the only information we need to process. As before, we can therefore group several graphs sharing the same span and attachment vertices into an equivalence class represented as a partial analysis, which we denote by a tuple of the form $[i_1, i_2, \ldots, i_q; h_1, h_2, \ldots, h_d]$. Figure 2(a) graphically represents a partial analysis $[i_1, i_2, i_3, i_4, i_5, i_6; h_1, h_2]$. At the heart of the parsing algorithm we have a set of binary combinations of partial analyses, resulting in new partial analyses for larger graphs. In Figure 2(b) we show one such operation, where partial analyses $[i_1, i_2, i_4, i_5; h_1, h_2]$ and $[i_3, i_4, i_6, i_7; h_3]$ are combined. Observe that the spans of these partial analyses share the index i_4. This means that, within w, some string in the span of the first partial analysis is "adjacent" to some string in the span of the second one. The operation creates two new dependency arcs, one with head h_3 and dependent h_1, and the other with head h_2 and dependent h_3. This means that the

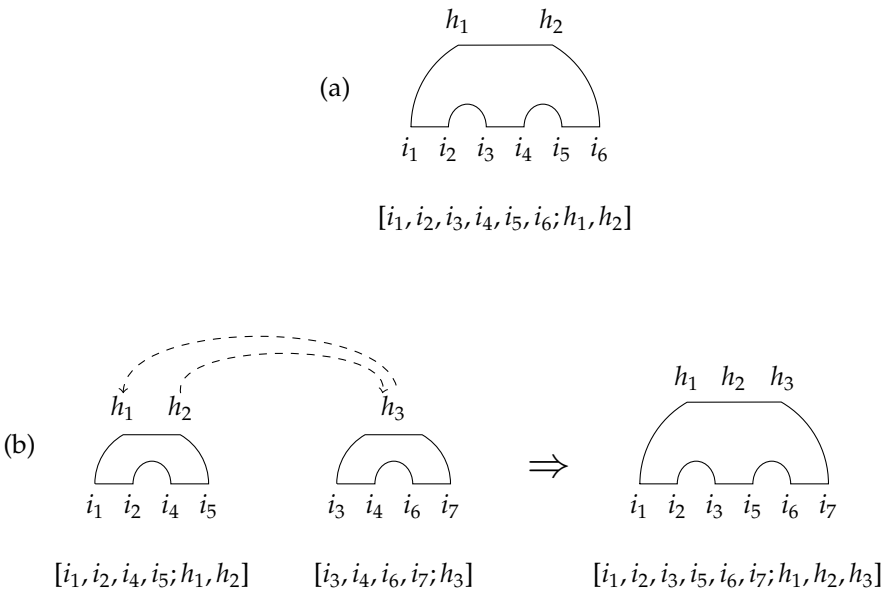


Figure 2
Graphical representation for (a) the partial analysis $[i_1, i_2, i_3, i_4, i_5, i_6, ; h_1, h_2]$ and (b) the combination of partial analyses $[i_1, i_2, i_4, i_5; h_1, h_2]$ and $[i_3, i_4, i_6, i_7; h_3]$ with $i_1 < i_2 < i_3 < i_4 < i_5 < i_6 < i_7$, producing partial analysis $[i_1, i_2, i_3, i_5, i_6, i_7; h_1, h_2, h_3]$.

underlying graph grammar is more powerful than a first-order model: see Section 1.1 for a precise definition of this terminology. The result of the combination is the partial analysis $[i_1, i_2, i_3, i_5, i_6, i_7; h_1, h_2, h_3]$. The span of the resulting analysis is obtained by merging the spans of the input analyses. In this specific case, the attachment vertices of the resulting analysis is the union of the attachment vertices of the input analyses: This means that each of these vertices is still seeking connections to vertices in w that are outside of the current span.

We now briefly analyze the efficiency of the string-to-graph parsing algorithm. Assume that q is the maximum number of indices in a span of our graphs, and d is the maximum number of attachment vertices. Again, each index i_k and $h_{k'}$ can range from 0 to n, the length of w. This means that the total number of partial analyses that need to be stored is in $\mathcal{O}(n^{q+d})$. As for the time complexity, let us assume that graph combination applies to two partial analyses whose spans share at least one index. We also assume that d_r is the maximum number of attachment vertices involved in a graph combining operation. Note that we must have $d_r \leq 2d$, since our operations are always binary. We then derive that the overall time spent by the algorithm for graph combination is in $\mathcal{O}(n^{2q-1+d_r})$. We observe that these space and time bounds are a generalization of the same bounds already discussed for string-to-tree parsing, where we have $q = 2$, $d = 1$, and $d_r = 2$, leading to $\mathcal{O}(n^3)$ space and $\mathcal{O}(n^5)$ time.

From this computational analysis we realize that if the grammar does not impose any constant bounds on the parameters q, d and d_r, then these values can grow with n and we end up with an exponential parsing algorithm, both in time and space. On the other hand, whereas constant bounds on these parameters result in a polynomial parsing algorithm, this comes at the cost of some loss in coverage, since some graphs might no longer be parsable. The choice of bounds for q, d, and d_r is essentially an empirical matter, in the sense that one should test the coverage of the restricted algorithm on a data set representative of the application domain. Furthermore, we note that there is a trade-off between q and d (and between q and d_r): one can achieve the desired coverage by setting a low value of q and increasing d, or else by setting a low value of d and increasing q.

To conclude our example, we have outlined that, when extending existing dynamic programming algorithms for string-to-tree parsing to string-to-graph parsing, we observe a huge gap in time and space efficiency. This gap does not depend on the specific class of formal grammars used to generate our structures, and is essentially due to any one of the two features that distinguish the two structures, namely, crossing arcs and multi-parenthood. We have also shown that one can improve the efficiency of string-to-graph parsing by restricting the class of graphs that can be parsed—that is, at the cost of some loss in coverage.

Before turning to a summary of the contributions of this article, we need to make one final step and introduce the grammar framework that we use to model semantic representations for natural language sentences, as well as the problem of extracting these grammars from data. We consider the class of **hyperedge replacement grammar**, or HRG (Drewes, Kreowski, and Habel 1997). HRGs generate graph languages in a way that generalizes traditional CFGs for string rewriting. More precisely, each rule in an HRG replaces a left-hand side nonterminal in some hosting graph by means of a right-hand side graph. Similarly to CFGs, rewriting is done in a way that depends only on the nonterminal in the left-hand side of the rule, and not on the content of the hosting graph. The specific way in which, after rewriting, the right-hand side graph is connected to the hosting graph depends on the technical notion of hyperedge, which will be discussed in detail later in Section 5.1.

In semantic parsing applications, HRGs are extracted from a large data set of semantic graphs. The process of grammar extraction is an *unsupervised* task, because a graph does not provide any information about the rules of the underlying grammar that have been used to generate the graph itself. This is in contrast with the extraction of CFGs from a tree bank, where each tree provides information about its derivation process. A known technique to extract HRG rules from a given graph is to construct a so-called **tree decomposition** of the graph itself. This is a tree structure where each node contains some vertices of the graph, and the same vertex can appear at different nodes; we provide a mathematical definition of tree decomposition in Section 2. In the context of this article, the important property of a tree decomposition is that each node can be converted into an HRG rule manipulating the vertices of the graph that appear in the node itself. From a tree decomposition we can then directly read off the rules of an HRG that generates the given graph. We say that a tree decomposition is optimal if it keeps to a minimum the maximum number of vertices contained in its nodes. It follows that an optimal tree decomposition keeps the size of the rules of the extracted HRG as small as possible and, as we will see, optimizes also the computational complexity of parsing algorithms based on HRGs. We can therefore reformulate our unsupervised learning problem of extracting an HRG from a graph as the problem of finding an optimal tree decomposition of a graph.

In the graph theory literature, a considerable amount of work has been devoted to the problem of finding optimal tree decompositions of a graph. In the general case, this is an NP-hard problem; see Section 1.1 for references. The novel idea in this article is to impose restrictions on tree decompositions, on the basis of the fact that the vertices of the input graph are associated with a total ordering. As we will see, this leads to some interesting classes of tree decompositions that can be optimized in polynomial time. Furthermore, optimal tree decompositions on available semantic data sets provide HRGs leading to algorithms for the string-to-graph problem running in polynomial space and time.

We are now ready to list the main contributions reported in this article.

- We define two novel, specialized tree decompositions for graphs associated with a fixed vertex ordering, called **inside** tree decompositions and **outside** tree decompositions. We provide algorithms that compute optimal inside and outside tree decompositions in polynomial time in the size of the input graphs.
- The HRG extracted from an optimal inside or outside tree decomposition, called inside or outside HRG, respectively, has the following properties (we use the notation of Example 1). Each generated subgraph can be represented by a partial analysis with $q = 2$. The value of d_r is as small as possible on the training data set, given the constraint that $q = 2$.
- We develop a dynamic programming algorithm for string-to-graph parsing based on HRGs, running in time $\mathcal{O}(|G|n^{3+d_r})$, where $|G|$ is the size of the input HRG, n is the number of input vertices, and d_r is specified as in the item above. This is a polynomial algorithm whenever d_r is bounded by a constant.
- We provide empirical assessment on a semantic data set annotated accordingly to the linguistic formalism called Abstract Meaning Representation, described by Banarescu et al. (2013). This shows that

inside and outside HRGs with $d_r = 5$ cover 99% of the sentences in the data set. On average, we have $d_r = 1.97$ for inside HRG and $d = 1.81$ for outside HRG, suggesting that for English most of the partial analyses can be stored and processed efficiently.

We close this introduction with a summary of the content of the remaining sections. After summarizing related work, we define tree decompositions and our notion of parse trees over vertex sequences in Section 2. In Section 3, we define our first family of tree decompositions, which we call inside tree decompositions, and give an algorithm for finding optimal inside tree decompositions. As our second family, we define outside tree decompositions and give a corresponding optimization algorithm in Section 4. Section 5 describes how to extract an HRG from a tree decomposition of either family, and how to parse a vertex sequence into a graph using an HRG. Section 6 provides the results of our grammar extraction algorithms on linguistic data sets, and Section 7 concludes with discussion and directions for future work.

1.1 Related Work

Hyperedge replacement grammars (Drewes, Kreowski, and Habel 1997) are among the most successful generative models for graph languages, because of their close relationship to the well-known class of CFGs. In the context of natural language processing, HRGs have been exploited to model semantic representations by Jones et al. (2012), who propose a synchronous version of HRG for machine translation, and by Jones, Goldwater, and Johnson (2013) and Peng, Song, and Gildea (2015), who propose algorithms for automatic learning of HRGs that are evaluated on the task of extracting semantic dependency graphs from text.

Considering the already mentioned graph parsing problem, where the input consists of a graph and a graph-rewriting grammar, one of the first algorithms for parsing based on HRG has been proposed by Lautemann (1990). For general HRGs, this algorithm runs in exponential time. This complexity result comes as no surprise, since it is known that graph parsing for HRG is an NP-hard problem, even for fixed grammars (Aalbersberg, Rozenberg, and Ehrenfeucht 1986; Lange and Welzl 1987). In the context of natural language processing, Chiang et al. (2013) proposed an optimized version of Lautemann, also providing a fine-grained complexity analysis that is missing in the original article. The running time of the optimized algorithm is an exponential function of both the treewidth of the input graph (to be defined in Section 2) and of the maximum degree of its nodes.

Polynomial time parsing algorithms for subclasses of HRG have also been investigated in the graph grammar literature. A predictive top–down parsing algorithm has been presented by Drewes, Hoffmann, and Minas (2015), inspired by the LL(1) parsing method for CFGs, and working for a restricted class of HRG. The algorithm runs in quadratic time in the size of the input graph. However, in the worst case, the constructed grammar table used to drive the parser has size exponential in the size of the input grammar. Björklund, Drewes, and Ericson (2016) also propose a quadratic time parsing algorithm, for a restricted class of HRG generating directed *acyclic* graphs, and discuss the relevance of this subclass for parsing of natural language semantic representations.

The term string-to-graph parsing has been introduced in this article to denote the problem of parsing a sequence of vertices into a graph using a graph grammar, in contrast to the graph parsing problem discussed earlier. We have already argued that these two problems are incomparable and that, in the context of semantic analysis of natural

language, string-to-graph parsing is more relevant than graph parsing. Nonetheless, all of the algorithms for natural language semantic analysis cited here solve the graph parsing problem, that is, they take as input a completely specified graph, and we are not aware of any published algorithm for string-to-graph parsing making use of a graph rewriting grammar.

String-to-graph parsing is very closely related to the so-called problem of **dependency semantic parsing**, where one is given an ordered sequence of vertices and has to output a maximum weight graph defined over those vertices, representing a most-likely semantic analysis of the associated sentence. In contrast with string-to-graph parsing, dependency semantic parsing does not require any hardwired input grammar generating a set of valid graphs. In practice, in all of the approaches to dependency semantic parsing we are aware of, the search space basically consists of every possible graph, and the weight of each graph is defined using some general schema that provides weights for elementary patterns appearing in the graph. More precisely, a pattern is either a single arc, called a first order pattern, or else a more complex combination of $k > 1$ arcs, called a k-th order pattern. The weight of a graph G is then computed by summing up the weight of each occurrence of a pattern in G, where different occurrences might overlap in case of patterns of order higher than 1. In this respect, we may therefore view semantic dependency parsing as the *grammarless* version of string-to-graph parsing, meaning that in the definition of the former problem the search space need not be defined by a generative grammar. Dependency semantic parsing has been a prominent shared task at recent editions of the International Workshops on Semantic Evaluation (SemEval). See Oepen et al. (2014) and Oepen et al. (2015) for a quick overview of the different approaches that have been evaluated.

Two grammarless algorithms for dependency semantic parsing are worth discussing here, because they are related to the dynamic programming approach that we use in Section 5 to solve string-to-graph parsing for HRG. Kuhlmann and Jonsson (2015) and Schluter (2015) have independently derived essentially the same algorithm for finding a maximum weight *projective* directed acyclic graph, given an input vertex sequence w and using a first-order model (a model using only first order patterns). The term projective is a generalization to graphs of the same concept as used in dependency trees: Informally, a projective graph does not have two arcs that cross each other, with respect to the ordering in w. The algorithm is based on the already mentioned CKY algorithm for text-to-string parsing under a CFG (Aho and Ullman 1972). It explores projective graphs whose vertices span a substring of w, and it does so by merging in some specific ways subgraphs with spans that are "adjacent" in w. At the top of the merge operations, a Viterbi search is performed to find the maximum weight graph(s). The algorithm runs in cubic time in the length of w. Using the notation of Example 1, this algorithm uses $q = 2$, $d = 2$, and $d_r = 4$, but with the restriction that the attachment vertices of each subgraph are the same as the vertices at the left and right boundaries of the span of the subgraph itself. This condition rules out crossing arcs and is only possible because of the projective restriction on the processed graphs. If we ignore the fact that Kuhlmann and Johnsson and Schluter use a grammarless approach, the parsing algorithm we present in this article can be viewed as a generalization of the work by those authors, since we relax the condition $d = 2$ and we do not impose any restriction on the position of the attachment vertices.

Other grammarless algorithms for dependency semantic parsing have been reported by Flanigan et al. (2014), who use maximum weight spanning techniques, and by Damonte, Cohen, and Satta (2017) and Gildea, Satta, and Peng (2018), who use special transition systems combined with greedy methods.

The connection between tree decomposition of a graph and HRG rules, which we use in this article, was first made by Lautemann (1988). In the context of natural language processing, the same idea has been previously exploited by Jones, Goldwater, and Johnson (2013), who introduce a number of heuristics to find tree decompositions of semantic graphs of natural language sentences. They extract corresponding HRG rules and analyze the characteristics of the resulting grammars. Unlike the methods we present in this article, the methods of Jones, Goldwater, and Johnson do not refer to the natural language string corresponding to the semantic graph, and therefore are not applicable to string-to-graph parsing.

In this article, we experiment with semantic data sets annotated according to the linguistic formalism called Abstract Meaning Representation, described by Banarescu et al. (2013). A general discussion of alternative formalisms and linguistic graph banks can be found in Kuhlmann and Oepen (2016).

2. Tree Decompositions and Parse Trees

The graphs that we use in this article have directed arcs, because this is a standard requirement for semantic representation of natural language sentences. We denote a directed graph as $G = (V, E)$, where V is the set of vertices and E is the set of edges—that is, ordered pairs of the form (v, u) with $v, u \in V$. A tree decomposition of G is a special tree where each node is associated with a subset of V. Because a tree is a particular kind of graph, to avoid confusion between the two we adopt the following convention: When describing tree decompositions we use the terms *node* and *arc*, and when describing graphs we use the terms *vertex* and *edge*.

Definition 1
Let $G = (V, E)$ be a graph. A **tree decomposition** of G is a tree $T = (N, A)$, where N is a set of nodes and A is a set of directed arcs. Each node $n \in N$ is associated with a set $Bag(n) \subseteq V$, referred to as a **bag**, in such a way that the following properties are all satisfied.

1. **Vertex cover**: The nodes of the tree T cover all the vertices of G: $\bigcup_{n \in N} Bag(n) = V$.
2. **Edge cover**: Each edge in G is included in some node of T. That is, for all edges $(u, v) \in E$, there exists an $n \in N$ with $u, v \in Bag(n)$.
3. **Running intersection**: The nodes of T containing a given vertex of G form a connected subtree of T. That is, for all $n, n_1, n_2 \in N$, if n is on the (unique) path in T from n_1 to n_2, then $Bag(n_1) \bigcap Bag(n_2) \subseteq Bag(n)$.

The **width** of a tree decomposition T is $\max_{n \in N} |bag(n)| - 1$. Let $TD(G)$ be the set of all valid tree decompositions of a graph G. The **treewidth** of G is the minimum width over all tree decompositions of G:

$$\mathbf{tw}(G) = \min_{\substack{T \in TD(G) \\ T = (N, A)}} \max_{n \in N} |bag(n)| - 1 \,.$$

We say that a tree decomposition T of G is **optimal** if its width equals $\mathbf{tw}(G)$.

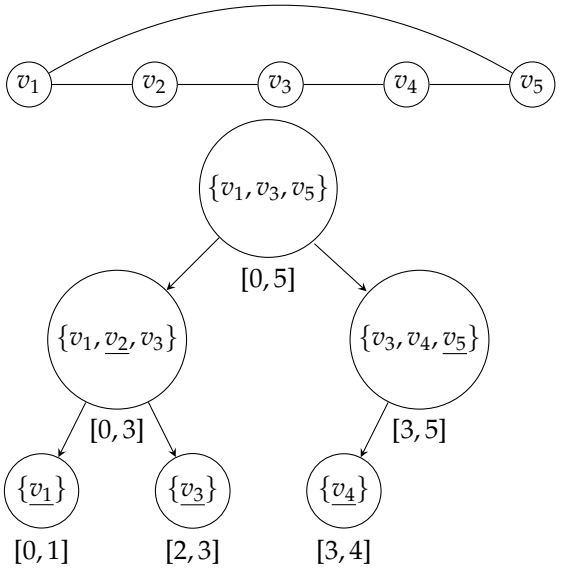


Figure 3
Graph G (top) consisting of a single cycle and the vertex ordering $v_1v_2v_3v_4v_5$. A possible tree decomposition T (bottom) for G, which is also a parse tree for w. Bags and spans are depicted inside and below circles, respectively. In each bag, we underline the vertex that is introduced at the corresponding node of T.

Example 2
A graph G is shown in Figure 3. More precisely, G has vertices $V = \{v_i \mid 1 \leq i \leq 5\}$ and edges $E = \{(v_i, v_{i+1}) \mid 1 \leq i \leq 4\} \cup \{(v_1, v_5)\}$. Figure 3 also presents a possible tree decomposition T for G. Inside each node of T we display the corresponding bag. It is easy to verify that the vertex cover and the edge cover properties from Definition 1 are both satisfied for T. As an example of the running intersection property, consider vertex v_3. This vertex appears in the bags of the nodes of T with labels $[2,3]$, $[0,3]$, $[0,5]$ and $[3,5]$, which form a connected subtree of T. The largest size of a bag for T is 3, and therefore the width of T is 2. It is possible to show that T is also an optimal tree decomposition—that is, no tree decomposition of G has width smaller than 2.

Definition 1 can apply to either a directed or an undirected graph G: Whether a tree decomposition is valid does not depend on the direction of the edges of G. Although a tree decomposition is typically an undirected tree in the literature on tree decomposition, in this article we will consider rooted trees with directed arcs. Furthermore, our tree decompositions are ordered trees, meaning that there is a total ordering for the children at any internal node. The direction and the ordering do not affect whether T is a valid tree decomposition of G, but will be useful in describing our algorithms.

The tree decompositions we investigate in this article are all relativized to some ordering w of the vertices of G that is provided as input. More specifically, we impose that a tree decomposition T is a *parse tree* for w. Before presenting the mathematical definition of parse tree, we informally introduce the basic idea. Let $w = v_1 \cdots v_{|V|}$. For each v_i, there is a unique, designated node of T that is said to *introduce* v_i, and each node can introduce at most one v_i. The nodes of T that introduce some v_i are called anchored, and the remaining nodes are called unanchored. We also require that internal nodes of T have at most two children. Recall that T is an ordered tree, so that at each internal

node we may distinguish between left and right children. If T is a parse tree for w then, when we perform an in-order traversal of T and print v_i whenever we visit the anchored node that introduces v_i, we obtain w.

We are now ready to introduce the mathematical definition of parse tree. For each i, j with $0 \leq i < j \leq |V|$, we write $[i, j]$ to denote the set of vertices $\{v_{i+1}, \ldots, v_j\}$. The pair $[i, j]$ is called a **span** of w.

Definition 2
Let $w = v_1 \cdots v_{|V|}$. A **parse tree** for w is a tree where each node is associated with some span $[i, j]$ of w, and T's root has span $[0, |V|]$. Each node of T is either an anchored or an unanchored node, specified as follows.

1. An **unanchored** node has span $[i, j]$ with $j - i \geq 2$, and has left and right children with spans $[i, k]$ and $[k, j]$, respectively, for some k with $i < k < j$.
2. An **anchored** node n with span $[i, j]$ **introduces** a unique vertex $v_k \in [i, j]$. We distinguish four cases, based on the position of v_k in the substring $v_{i+1} \cdots v_j$ of w.
   (a) If $j - i = 1$, then n is a leaf and $k = j$.
   (b) If $j - i \geq 2$ and $k = i + 1$, then n has only a right child, with span $[i + 1, j]$.
   (c) If $j - i \geq 2$ and $k = j$, then n has only a left child, with span $[i, j - 1]$.
   (d) If $j - i \geq 3$ and $k \notin \{i + 1, j\}$, then n has left and right children with spans $[i, k - 1]$ and $[k, j]$, respectively, for some k with $i + 1 < k < j$.

According to Definition 2, the span at each internal node n of T is always the union of the following disjoint sets: the spans at the children (or at the single child) of n and, if n is anchored, the set $\{v\}$ where v is the vertex introduced at n. We also observe that each vertex is introduced by exactly one node of T, and each node introduces at most one vertex.

Example 3
Consider again the graph G and the tree decomposition T in Figure 3. Tree T is also a parse tree for the vertex ordering $w = v_1 v_2 v_3 v_4 v_5$, with both anchored and unanchored nodes. When we underline a vertex v_i in $Bag(n)$, it means that n is an anchored node introducing v.

The span of each node is also depicted at the bottom of the node itself. For instance, the node with span $[0, 5]$ is a binary unanchored node with left and right children having span $[0, 3]$ and $[3, 5]$, respectively. The node with span $[3, 5]$ is a unary anchored node, introducing vertex v_5 and with left child having span $[3, 4]$. Finally, the node with span $[0, 3]$ is a binary anchored node introducing vertex v_2 and with left and right children having span $[0, 1]$ and $[2, 3]$, respectively. The span $[0, 3]$ is the union of the disjoint spans at the children and the set $\{v_2\}$ containing the introduced vertex.

If we perform an in-order traversal of T and print the introduced vertices at each anchored node, we obtain w. This fact is a general property of parse trees, and will be proven below.

To help the reader's intuition, we observe that there is a strong similarity between parse trees and trees generated by a CFG whose rules are of the form

$$S \to SS, \quad S \to SvS, \quad S \to vS, \quad S \to Sv, \quad S \to v,$$

where S is a nonterminal symbol and v is a terminal symbol denoting any of the vertices in w. The only difference is that in a parse tree T each vertex v is introduced by some node n rather than being attached as a child node of n, as in the previous grammar.

Consider now a node n of T with span $[i, j]$, and let T_n be the subtree of T rooted at n. We show by induction that an in-order traversal of T_n printing v_i whenever we visit an anchored node that introduces v_i, results in the substring $v_{i+1} \cdots v_j$ of w. In the base case n is a leaf node, and we have $j - i = 1$. According to Definition 2, n introduces v_j, and therefore an in-order traversal of T_n prints the substring $v_{i+1} \cdots v_j = v_j$. Assume now that n is an anchored binary node, with left child n_L having span $[i, k-1]$ and right child n_R having span $[k, j]$, for some k with $i + 1 < k < j$. According to Definition 2, node n introduces vertex v_k. By the inductive hypothesis an in-order traversal of T_{n_L} prints $v_{i+1} \cdots v_{k-1}$ and an in-order traversal of T_{n_R} prints $v_{k+1} \cdots v_j$. We then have that an in-order traversal of T_n prints $v_{i+1} \cdots v_{k-1} \cdot v_k \cdot v_{k+1} \cdots v_j = v_{i+1} \cdots v_j$. The remaining cases for an internal node n can be dealt with in a similar way. Finally, from this property it follows that the set of vertices introduced by the anchored nodes within T_n is $[i, j]$, since $[i, j]$ contains all and only the symbols in $v_{i+1} \cdots v_j$.

Definition 2 places constraints on the tree structure of the tree decomposition T, based on the ordering w, but does not refer to the contents of the bag at each node of T. Thus, the problem of finding a tree decomposition of G can be broken into the two problems of, first, finding a parse tree for w and, second, identifying the vertices of G to include in the bag at each node of T.

Notation. Throughout this article we use the following general notation for the vertices in V and the nodes of T. For $v \in V$, we write $N(v)$ to denote the set of all neighbors of v; formally, $N(v) = \{u \mid (v, u) \in E \text{ or } (u, v) \in E\}$. We also write $T(v)$ to denote the anchored node of T that introduces v. For each $u \in N(v)$, we write $T(v, u)$ to denote the lowest common ancestor in T of the anchored nodes $T(v)$ and $T(u)$. Because each $T(v, u)$, $u \in N(v)$, dominates $T(v)$, nodes $T(v, u)$ are all in a path from $T(v)$ to T's root. We then write $T(v, N(v))$ to denote the highest node in such a path among all the nodes $T(v, u)$, $u \in N(v)$.

3. Inside Tree Decomposition

Assume a graph $G = (V, E)$, with $V = \{v_1, \ldots, v_{|V|}\}$, and let $v_1 \cdots v_{|V|}$ be a string representing an ordering of the vertices in V. In this section we introduce our first family of tree decompositions, called inside tree decompositions, and show how to compute optimal tree decompositions for this family. We use the notation $T(v)$, $T(v, v')$, and $T(v, N(v))$ introduced at the end of Section 2.

For our inside tree decompositions, we use a restricted version of the parse trees in Definition 2. An **inside parse tree** is a parse tree that has anchored nodes only at its leaves. Because internal nodes of an inside parse tree are all unanchored, there are only binary nodes in these trees. Furthermore, in an inside parse tree the vertex ordering $v_1 \cdots v_{|V|}$ can be directly obtained from the yield of the tree itself.

Definition 3
A tree decomposition T of G is an **inside tree decomposition** of G relative to vertex order $v_1 \cdots v_{|V|}$ if the following conditions are both satisfied.

1. Tree T is an inside parse tree for $v_1 \cdots v_{|V|}$.
2. Each vertex $v_i \in V$ appears in the bags at nodes $T(v_i, N(v_i))$ and $T(v_i)$, and at each node on the path between these two nodes.

Example 4
Consider again the graph G of Example 2, repeated for convenience at the top of Figure 4, and the associated vertex ordering $v_1v_2v_3v_4v_5$. There are several inside tree decompositions of G, relative to $v_1v_2 \cdots v_5$, depending on the specific choice for the topology of the underlying inside parse tree for $v_1v_2 \cdots v_5$. One possible inside decomposition T is shown at the bottom of Figure 4. Inside each node of T we display the

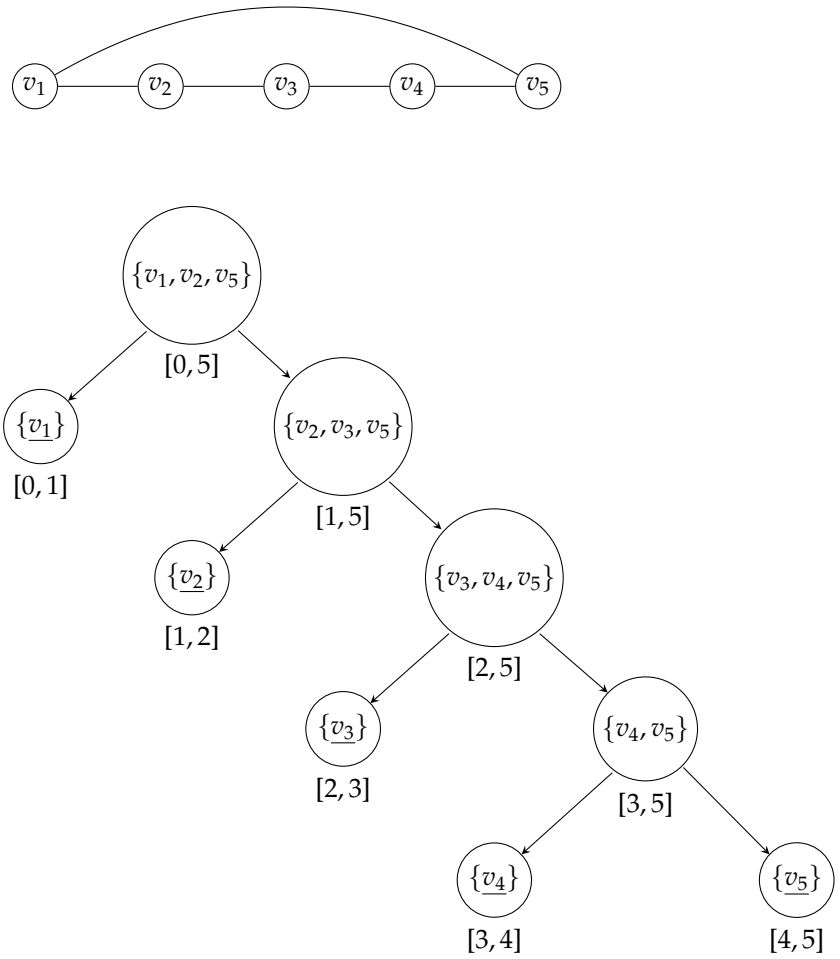


Figure 4
Graph G (top) consisting of a single cycle and the vertex ordering $v_1v_2v_3v_4v_5$. A possible inside tree decomposition T (bottom) for G and the given ordering. Bags and spans are depicted inside and below circles, respectively.

corresponding bag. Anchored nodes appear all at the leaves of T. Following Example 3, we underline the vertex in the bag that is introduced at each anchored node, even if there is no ambiguity in this case, since each leaf has only one vertex in its bag.

Consider for instance vertex v_3 with set of neighbors $N(v_3) = \{v_2, v_4\}$. Node $T(v_3)$ introduces v_3 and has span $[2,3]$, node $T(v_3, v_2)$ has span $[1,5]$, and node $T(v_3, v_4)$ has span $[2,5]$. Then $T(v_3, N(v_3)) = T(v_3, v_2)$, and v_3 appears in the bags of all and only the nodes in the path from $T(v_3)$ to $T(v_3, v_2)$.

More interestingly, consider vertex v_5 with $N(v_5) = \{v_4, v_1\}$, and observe that v_1 is not adjacent to v_5 in the given vertex order. Node $T(v_5)$ introduces v_5 and has span $[4,5]$, node $T(v_5, v_4)$ has span $[3,5]$, and node $T(v_5, v_1)$ has span $[0,5]$. Then $T(v_5, N(v_5)) = T(v_5, v_1)$, and v_5 appears in the bags of all and only the nodes in the path from $T(v_5)$ to $T(v_5, v_1)$.

We now show that any tree T specified as in Definition 3 is a valid tree decomposition of G, according to Definition 1. First, each vertex $v_i \in V$ is included in $Bag(T(v_i))$, satisfying the vertex cover condition. For each edge $(v, v') \in E$, vertices v and v' are both included in $Bag(T(v, v'))$, satisfying the edge cover condition. Finally, for each vertex $v_i \in V$, all the nodes n of T such that $v_i \in Bag(n)$ form a subtree of T that is simply the path starting at $T(v_i, N(v_i))$ and ending with $T(v_i)$. This satisfies the running intersection condition.

Inside tree decompositions cannot always achieve optimal treewidth. This means that there exist graphs G such that, regardless of the vertex order, no inside tree decomposition of G achieves the width $\mathbf{tw}(G)$ of an optimal tree decomposition of G. We will show this by means of an example.

Example 5

In this example we ignore edge directions in graphs as well as arc directions in tree decompositions, because this information is not relevant in this case. Figure 5 shows a graph G and an optimal tree decomposition T of width 2. We show here that no inside tree decomposition of G can achieve width 2.

We first show that any optimal tree decomposition must contain the bags shown in Figure 5, arranged in the same general pattern. A **clique** is a set of vertices in a graph with an edge joining each pair of vertices in the set. For each clique W in G, and for each tree decomposition T' of G, there must be a bag in T' containing the vertices in W; this is a well-known property of tree decompositions. To be optimal, T' must have no bags of more than three vertices. Hence, each bag in T appears as a bag of T'. Consider now the three leaf bags in T. It can be easily verified that, for each of these bags, the path in T' to the bag 123 must not pass through any of the other leaf bags of T, because this would violate running intersection. Hence, T' must contain the basic structure shown in T, possibly with additional bags interposed along the edges shown in T.

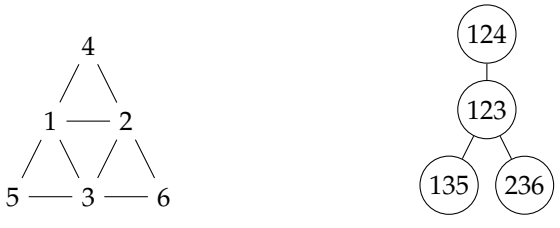


Figure 5
Left: A graph G for which no inside tree decomposition is optimal. Right: An optimal tree decomposition for G. Note that, regardless of which bag is chosen as the root, there exists a vertex v such that the bags containing v are not all on a path from the root to a leaf.

An inside tree decomposition is a rooted tree, and thus one of the bags shown in T must appear closest to the root. By considering each of the four possibilities, one can verify that, regardless of which bag is chosen as closest to the root, some vertex is contained in bags in a path which travels up one branch of the tree and then down another branch. But, in an inside tree decomposition, a vertex can only appear along a path from one node up toward the root. Thus, the structure shown in T cannot appear in an inside tree decomposition, and thus no inside tree decomposition can achieve the optimal width $\mathbf{tw}(G)$.

We now consider the problem of taking as input a graph $G = (V, E)$ and some vertex order $v_1 \cdots v_{|V|}$, and finding the optimal inside tree decomposition for G relative to the given order. This is the inside tree decomposition having its largest bags with as few vertices as possible. The problem is not trivial, because there is a valid inside tree decomposition corresponding to any binary tree over the input, and the number of such trees is not bounded by any polynomial in $|V|$. This problem can be broken into two tasks: constructing the forest with all the skeleton structures of the tree decompositions, and performing a min-max search on the entire forest. We provide a dynamic programming algorithm that intermixes these two tasks. The algorithm is based on the CKY algorithm for parsing using CFGs (Aho and Ullman 1972).

We start by developing a characterization of the sets $Bag(n)$. Recall that $N(v)$ is the set of all neighbors of v. We write $I(i, j)$ to denote the set of vertices in $[i, j]$ that have neighbors outside of $[i, j]$:

$$I(i, j) = \{v \mid v \in [i, j],\ u \in N(v),\ u \notin [i, j]\}$$

Theorem 1
Let T be an inside tree decomposition of G relative to $v_1 \cdots v_{|V|}$, and let n be a node of T with span $[i, j]$. Then the following conditions must hold.

1. If n is a leaf node introducing vertex v_j, which implies $i = j - 1$, then $Bag(n) = \{v_j\}$.
2. If n is an internal node whose children have spans $[i, k]$ and $[k, j]$, respectively, then $Bag(n) = I(i, k) \cup I(k, j)$.

Conversely, any inside parse tree for $v_1 \cdots v_{|V|}$ meeting Conditions 1 and 2 at each node is an inside tree decomposition of G relative to $v_1 \cdots v_{|V|}$.

Proof. We start with the first part of the statement of the theorem, and assume n is a node of T with span $[i, j]$.

Condition 1. Suppose $n = T(v_j)$, that is, n is the leaf node introducing v_j and $i = j - 1$. It follows directly from Definition 3 that $Bag(n) = \{v_j\}$.

Condition 2. Suppose n is an internal node whose children have spans $[i, k]$ and $[k, j]$, respectively. We first show $I(i, k) \subseteq Bag(n)$. Let n_L be the left child of n, which has span $[i, k]$. Consider an arbitrary vertex $v \in I(i, k)$. By the definition of $I(i, k)$, there exists a neighbor $u \in N(v)$ that is outside of $[i, k]$. Then the least common ancestor of $T(v)$ and $T(u)$, $T(v, u)$, dominates n_L, because n_L dominates $T(v)$ but not $T(u)$. Node $T(v, N(v))$ also dominates n_L, and therefore v appears in $Bag(n)$ by Definition 3. Because v was arbitrarily chosen in $I(i, k)$, we conclude that $I(i, k) \subseteq Bag(n)$. By the same reasoning, $I(k, j) \subseteq Bag(n)$.

In the other direction, we show that all vertices in $Bag(n)$ appear in either $I(i,k)$ or $I(k,j)$. First we note that $Bag(n) \subseteq [i,j]$. To see this, observe that each vertex $v \in Bag(n)$ appears only in bags on a path from $T(v,N(v))$ to $T(v)$, where $T(v)$ has span $[k-1,k]$ for some k such that $[k-1,k] = \{v\} \subseteq [i,j]$. Furthermore, if $v \in [i,k]$ and if v has no neighbors outside $[i,k]$, then $T(v,N(v))$ will be below n, and v will not appear in $Bag(n)$. Thus v is in $Bag(n)$ only if v has a neighbor not in $[i,k]$, and hence $v \in I(i,k)$, by the definition of set $I(i,k)$. Similarly, if $v \in [k,j]$, v is in $Bag(n)$ only if v has a neighbor not in $[k,j]$, and hence $v \in I(k,j)$.

We now show the second part of the theorem. Let T be an inside parse tree. Conditions 1 and 2 specify a unique collection of bags for the nodes of T. We also observe that Condition 2 in Definition 3 applies to any inside parse tree for $v_1 \cdots v_{|V|}$, and therefore to T as well, producing a unique collection of bags for its nodes, and we have already shown that the collections of bags for the nodes of T produced by the two definitions are the same. ■

The important thing about Theorem 1 is that, in an inside tree decomposition, the bag of an internal node n depends only on the span of n's children. This means that, in order to compute $Bag(n)$, we do not need to store the bags at n's children. This fact is at the basis of our optimization method, which is reported in Algorithm 1.

Algorithm 1 computes quantities $b(i,j)$ for each i,j with $0 \leq i < j \leq |V|$, storing one plus the minimum width among all inside tree decompositions with root span $[i,j]$ (we work in terms of bag size, which is width plus one). As already mentioned, the algorithm follows the basic structure of the CKY chart parsing algorithm, adopting a bottom–up strategy, and combines it with a min-max search. At line 3, we initialize quantities $b(i-1,i)$ with the bag size of a leaf node, which is always one. At line 7, we compute the maximum bag size of an inside decomposition formed by a root node n with span $[i,j]$ and with two children with span $[i,k]$ and $[k,j]$, respectively, that are the roots of optimal inside decompositions. The maximum bag size is therefore the max among the same value at the children decompositions and the size of the bag at n, which only depends on the span of n's children, by Theorem 1. At line 8 we store the running minimum among the values computed at line 7. In this way, when we exit the for loop of line 6, quantity $b(i,j)$ stores the minimum width plus one among all decompositions with root span $[i,j]$. Finally, we return $b(0,|V|)-1$, which is the width of an optimal inside tree decomposition of G relative to $v_1 \cdots v_{|V|}$.

Algorithm 1 Computation of the width of an optimal inside tree decomposition of G relative to $v_1 \cdots v_{|V|}$.

```
1: procedure InsideTreeWidth(G = (V, E), v_1 ··· v_|V|)
2:     for each i do
3:         b(i − 1, i) ← 1
4:     for each i, j in ascending order of j − i > 1 do
5:         b(i, j) ← +∞
6:         for each k with i < k < j do
7:             b ← max{|I(i, k) ∪ I(k, j)|, b(i, k), b(k, j)}
8:             b(i, j) ← min{b(i, j), b}
9:     return b(0, |V|) − 1
```

Theorem 2
Algorithm 1 finds the width of the optimal inside tree decomposition of the input graph relative to the input vertex order $v_1 \cdots v_{|V|}$.

Proof. Let k be the result of Algorithm 1 and let k^* be the width of an optimal inside tree decomposition. To show that $k^* \leq k$, let T be the tree visited by the dynamic programming procedure of Algorithm 1 when applying the min operator at line 8. Tree T is a binary tree having leaves corresponding to the vertices $v_1 \cdots v_{|V|}$, and therefore forms an inside parse tree. By the converse part of Theorem 1, the bags assigned by Algorithm 1 form an inside tree decomposition, and Algorithm 1 returns the treewidth of this tree decomposition, which cannot be smaller than k^*.

To show that $k \leq k^*$, suppose that T^* is an inside tree decomposition having the optimal width. Each bag in T^* must have the contents assigned by Algorithm 1 by the first part of Theorem 1. By induction over nodes n in T^* in bottom–up order, with $[i, j]$ the span of n, each quantity $b(i, j)$ at the completion of the inner loop at line 6 can be no larger than the maximum bag size of the subtree of T^* rooted at n. Therefore the treewidth returned by Algorithm 1 can be no larger than the treewidth of T^*. ■

We now analyze the computational complexity of Algorithm 1, and show how to use this algorithm to retrieve optimal inside tree decompositions.

Theorem 3
An optimal inside tree decomposition of G relative to $v_1 \cdots v_{|V|}$ can be found in time $\mathcal{O}(|V|^3)$.

Proof. We start by showing how to efficiently compute quantities $I(i, j)$, which are taken for granted by Algorithm 1. For each $v \in V$, let $lm(v)$ be the leftmost neighbor of v, relative to $v_1 \cdots v_{|V|}$; assume $lm(v) = v$ in case v has no left neighbors. Similarly, $rm(v)$ is the rightmost neighbor of v. We can efficiently compute $lm(v)$ and $rm(v)$ as follows. Start with $lm(v) = rm(v) = v$ for each $v \in V$. For each $v \in V$ and each right neighbor u of v, if v is at the left of $lm(u)$ then update $lm(u)$ with v. Similarly, for each left neighbor u of v, if v is at the right of $rm(u)$ then update $rm(u)$ with v. This preprocessing can be carried out in time $\mathcal{O}(|V| + |E|)$.

Recall that $I(i, j)$ consists of vertices in $[i, j]$ with neighbors outside $[i, j]$. Thus $v \in [i, j]$ belongs to $I(i, j)$ if and only if either (1) v has a neighbor to the left of $[i, j]$, in which case $lm(v) = v_k$ with $k \leq i$, or (2) v has a neighbor to the right of $[i, j]$, in which case $rm(v) = v_k$ with $k > j$. This can be used to compute $I(i, j)$ for all i, j with $0 \leq i < j \leq |V|$ in time $\mathcal{O}(|V|^3)$.

Turning now to Algorithm 1, the loop structure results in $\mathcal{O}(|V|^3)$ executions of the innermost block at lines 7 and 8. Because $I(i, j) \subseteq [i, j]$, as already observed, at line 7 we have that $I(i, k)$ and $I(k, j)$ are disjoint sets, and we can write $|I(i, k) \cup I(k, j)| = |I(i, k)| + |I(k, j)|$. We can therefore compute this quantity in time $\mathcal{O}(1)$, if we have previously stored the size of each $I(i, j)$. All of the remaining operations at lines 7 and 8 can be executed in time $\mathcal{O}(1)$ as well. We thus conclude that Algorithm 1 can be implemented to run in time $\mathcal{O}(|V|^3)$.

In order to be able to retrieve an optimal inside decomposition, we can modify Algorithm 1 by saving so-called backpointers. These are the values of k at line 8 that correspond to the computed minimum for each $b(i, j)$. After the execution of Algorithm 1, we start at $b(0, |V|)$ and use the backpointers to retrieve the parse tree of an optimal

inside decomposition in time $\mathcal{O}(|V|)$. We can then annotate each node of the tree with the associated bag in time $\mathcal{O}(b(0, |V|))$ per node, using Theorem 1. We therefore conclude that, after the execution of Algorithm 1, the process of retrieving an optimal inside decomposition can be carried out in time $\mathcal{O}(|V| \cdot b(0, |V|))$, which is in $\mathcal{O}(|V|^2)$. ■

4. Outside Tree Decomposition

We introduce in this section an alternative tree decomposition, called outside, that is complementary to the inside tree decomposition of Section 3. Informally, the inside tree decomposition stores, at each bag, vertices inside the span that need to be connected to the portion of the graph that lies outside of the span itself. Complementary to this, the outside tree decomposition stores at each bag vertices outside the span that need to be connected to the portion of the graph that lies inside the span. In what follows, we assume the definitions of graph $G = (V, E)$, vertex order $v_1 \cdots v_{|V|}$, and span $[i, j]$ from Section 2.

Recall that $N(v)$ is the set of all neighbors of v. We write $O(i, j)$ to denote the set of neighbors of vertices in $[i, j]$ that are themselves outside the span:

$$O(i, j) = \{u \mid v \in [i, j],\ u \in N(v),\ u \notin [i, j]\}$$

We define an **outside parse tree** for $v_1 \cdots v_{|V|}$ to be a parse tree as in Definition 2 meeting the following additional restriction: For every vertex $v \in V$ such that $[i, j]$ is the span of $T(v)$, and for every vertex u such that $u \in N(v)$ and $u \in O(i, j)$, node $T(u)$ must dominate $T(v)$ in T. In words, we require that each neighbor u of v that is not introduced below node $T(v)$, must be introduced at some node of T that dominates $T(v)$. From this condition, it follows that for each $(v, u) \in E$, one node among $T(v)$ and $T(u)$ must dominate the other in T.

Example 6
Consider again the graph G of Example 2, repeated for convenience at the top of Figure 6, and the associated vertex ordering $v_1 v_2 v_3 v_4 v_5$. A possible outside parse tree for G and the given vertex order is displayed at the bottom of Figure 6. The span of each node is depicted to the right of the node itself. Inside each node we report the corresponding bag, which will be defined later and should be ignored for now, and we underline the vertex that is introduced at the node itself.

There are no unanchored nodes in this outside parse tree. For instance, the node with span $[0, 5]$ is a unary anchored node, introducing vertex v_1 and with right child the node with span $[1, 5]$. The latter node is a binary anchored node having left child with span $[1, 3]$ and right child with span $[4, 5]$, and introducing vertex v_4. Finally, one can check that, for each edge (v, u) of G, we have that either $T(v)$ dominates $T(u)$ or else $T(u)$ dominates $T(v)$.

Definition 4
A tree decomposition T of G is an **outside tree decomposition** of G relative to vertex order $v_1 \cdots v_{|V|}$ if the following conditions are both satisfied.

1. Tree T is an outside parse tree for $v_1 \cdots v_{|V|}$.
2. Each vertex $v_i \in V$ appears in the bags at nodes $T(v_i)$, $T(u)$, and at each node on the path between these two nodes, for every $u \in N(v_i)$ such that $T(v_i)$ dominates $T(u)$.

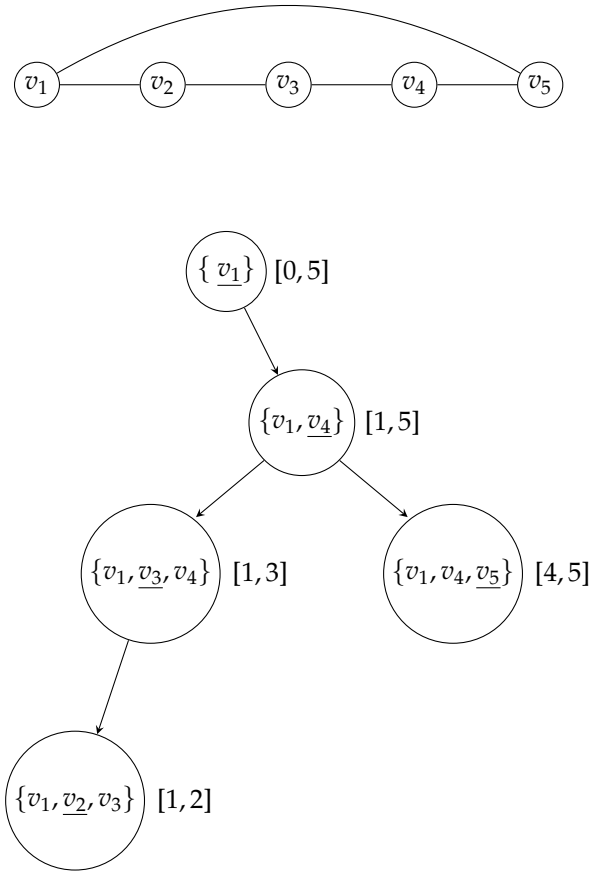


Figure 6
Graph G (top) from Example 2 with associated vertex ordering $v_1v_2v_3v_4v_5$. A possible outside tree decomposition T (bottom) for G and the given ordering. Bags and spans are depicted inside and to the right of circles, respectively. In each bag, we underline the vertex that is introduced at the corresponding node of T.

Example 7
Consider again the graph G from Figure 6. The outside parse tree T at the bottom of that figure is also an outside tree decomposition of G relative to vertex ordering $v_1v_2v_3v_4v_5$. To see this, consider for instance vertex v_1 of G, having neighbor vertices $N(v_1) = \{v_2, v_5\}$. $T(v_1)$ is the anchored unary node with span $[0, 5]$, $T(v_2)$ is the leaf node with span $[1, 2]$, and $T(v_5)$ is the leaf node with span $[4, 5]$. Because $T(v_1)$ dominates $T(v_2)$, v_1 appears in each bag in the downward path from $T(v_1)$ to $T(v_2)$. Similarly, because $T(v_1)$ dominates $T(v_5)$, v_1 appears in each bag in the downward path from $T(v_1)$ to $T(v_5)$.

We now show that any tree T satisfying Definition 4 is a valid tree decomposition according to Definition 1. We observe that each vertex v appears in $Bag(T(v))$, satisfying the vertex cover condition. Furthermore, for each edge $(u, v) \in E$, vertices u, v both belong to either $Bag(T(u))$ or $Bag(T(v))$, whichever is lower in the tree. This satisfies the edge cover condition. Finally, we need to show that the running intersection condition

is satisfied. Let v be an arbitrary vertex of G. According to Condition 2 in Definition 4, node $T(v)$ introducing v is the highest node in T with v in its bag. Furthermore, the nodes of T such that $v \in Bag(n)$ are placed in the paths from $T(v)$ to the nodes $T(u)$, for every $u \in N(v)$ such that $T(v)$ dominates $T(u)$. This means that the nodes of T with $v \in Bag(n)$ form a connected subtree of T rooted at $T(v)$ and with leaves the nodes $T(u)$ above.

As already observed in Example 5, the inside tree decompositions of Section 3 are sub-optimal, in the sense that, regardless of the vertex ordering, they might not be able to provide the smallest possible width for a given graph. In contrast, outside tree decompositions have this optimality property, as shown in the next theorem.

Theorem 4
Let G be some graph with vertex set V. Any tree decomposition of G can be transformed into an outside tree decomposition of G, relative to some ordering of V, preserving its width.

Proof. Let T be an arbitrary tree decomposition of G. Choose an arbitrary node of T as the root, add directions to each arc of T accordingly, and fix an arbitrary ordering for the children at each internal node of T. The resulting tree decomposition is called T_1. We specify now a few steps that, starting from T_1, construct new, intermediate decompositions T_2, T_3, T_4 of G that are rooted, directed, binary ordered trees.

We first need to introduce some auxiliary notation. Let v be an arbitrary vertex in V. For a tree decomposition T_i, we write $H(T_i, v)$ to denote the highest (closer to the root) node of T_i whose bag contains v. Node $H(T_i, v)$ is unique, since the running intersection condition in Definition 1 states that the nodes in T_i that contain v in their bags are all connected. Informally, we say that $H(T_i, v)$ contains the highest occurrence of v.

- For each internal node n of T_1 that has $q > 2$ children, we create $q - 2$ new nodes that are placed intermediately between n and its children in T_1, in such a way that n and the new nodes are all binary. The bag at each intermediate node is a copy of $Bag(n)$. The resulting tree T_2 is rooted and binary, and is still a valid tree decomposition of G.
- Assume there exists a node n of T_2 that contains the highest occurrences of vertices $v_1, \ldots, v_q$, $q \geq 2$, that is, we have $H(T_2, v_i) = H(T_2, v_j)$ for every i, j with $i \neq j$, $1 \leq i, j \leq q$. Then we replace n by means of a unary chain of q nodes $n_1, \ldots, n_q$, whose bags are copies of $Bag(n)$. Afterward, for each i with $1 \leq i \leq q$, we remove vertices $v_{i+1}, \ldots, v_q$ from n_i. If a unary node n of T_2 does not contain the highest occurrence of any vertex in V, then n must have a parent node n_p and $Bag(n) \subseteq Bag(n_p)$. We remove n and create a new arc joining n_p with the only child of n. Let T_3 be the tree obtained at this step. Note that T_3 is still a valid tree decomposition of G. This is because we have removed a vertex from bags only when the associated nodes were the highest positions in the tree containing the vertex, and had a single child containing the vertex. Furthermore, we have removed nodes from the tree only when the associated bags were included in the bag at the parent node. This preserves the three properties in the definition of tree decomposition (Definition 1). Finally, observe that each binary node in T_3

may contain the highest occurrence of at most one vertex, and each unary node contains the highest occurrence of exactly one vertex.

- We produce a tree T_4 by pruning T_3 as follows. Consider any pair of vertices u, v such that $u \in N(v)$ and such that $H(T_3, v)$ dominates $H(T_3, u)$. If v has no neighbor vertex in any of the bags below node $H(T_3, u)$, then we remove possible occurrences of v from the bags at each node below $H(T_3, u)$. We then prune all nodes of T_3 that have an empty bag. We call T_4 the resulting tree. Again, it is not difficult to see that T_4 is a valid tree decomposition of G.

We now argue that T_4 is an outside tree decomposition of G, relative to some vertex ordering. For any vertex $v \in V$, we let $H(T_4, v)$ be the anchored node introducing v. We also need to assign a span to each node of T_4. For each unary node n of T_4, we arbitrarily classify n's child as a left child or as a right child. We then perform an in-order visit of T_4, and print each vertex v when we encounter $H(T_4, v)$, that is, the node of T_4 that introduces v. Let $v_1 \cdots v_{|V|}$ be the resulting vertex ordering. For each node n of T_4, let $v_{i+1} \cdots v_j$ be the substring of $v_1 \cdots v_{|V|}$ that is printed while visiting the subtree of T_4 rooted at n. We associate n with the span $[i, j]$.

This span assignment satisfies the properties of the outside parse trees we use for outside tree decompositions. In particular, for every node $H(T_4, v)$ with span $[i, j]$ and for every vertex $u \in N(v)$ such that $u \in O(i, j)$, we have that $H(T_4, u)$ dominates $H(T_4, v)$. To see this, we observe that $H(T_4, u)$ cannot be dominated by $H(T_4, v)$, since $u \in O(i, j)$. Furthermore, if $H(T_4, u)$ does not dominate $H(T_4, v)$, then there would be no node n in T_4 such that $u, v \in Bag(n)$. Because v and u are joined by some edge of G, this violates the edge cover condition in Definition 1, against the fact that T_4 is a valid tree decomposition of G. ■

We now show a mathematical property that characterizes the family of outside tree decompositions. This property will be used later in the design of a dynamic programming algorithm that computes optimal outside tree decompositions of G.

Theorem 5
Let T be an outside tree decomposition of G relative to $v_1 \cdots v_{|V|}$, and let n be a node of T with span $[i, j]$. Then the following conditions must hold.

1. If n is an unanchored (binary) node whose children have spans $[i, k]$ and $[k, j]$, respectively, then $O(i, k) \cap [k, j] = \varnothing$, $[i, k] \cap O(k, j) = \varnothing$, and $Bag(n) = O(i, j)$.
2. If n is an anchored binary node whose children have spans $[i, k-1]$ and $[k, j]$, respectively, then $O(i, k-1) \cap [k, j] = \varnothing$, $[i, k-1] \cap O(k, j) = \varnothing$, and $Bag(n) = O(i, j) \cup \{v_k\}$.
3. If n is an anchored unary node or an anchored leaf node (with $j = i + 1$ if n is a leaf) introducing a vertex v, then $Bag(n) = O(i, j) \cup \{v\}$.

Conversely, any parse tree for $v_1 \cdots v_{|V|}$ with nodes meeting Conditions 1 to 3 is an outside parse tree and an outside tree decomposition of G relative to $v_1 \cdots v_{|V|}$.

Proof. We start with the first part of the statement of the theorem, and assume n is a node of T with span $[i, j]$. We separately prove Conditions 1 to 3.

Condition 1. Let n_L be the left child of n and n_R be the right child. Consider a vertex $u \in O(i, k)$. According to the definition of outside parse tree, $T(u)$ dominates n_L, so $T(u)$ is not in the subtree of T rooted at n_R and $u \notin [k, j]$. Thus $O(i, k) \cap [k, j] = \varnothing$. A similar argument shows that $[i, k] \cap O(k, j) = \varnothing$.

Consider now $u \in O(i, j)$. By the definition of $O(i, j)$, $u \notin [i, j]$ and there must exist a vertex $v \in [i, j]$ such that $u \in N(v)$. According to the definition of outside parse tree, node $T(v)$ must be either in the subtree rooted at n_L or in the subtree rooted at n_R, and $T(u)$ must dominate the root of that subtree. Then u appears in $Bag(n)$ by Definition 4. This shows that $O(i, j) \subseteq Bag(n)$.

In the other direction, let $u \in Bag(n)$. There must exist a vertex v such that $u \in N(v)$, with $T(v)$ below n and $T(u)$ dominating $T(v)$. Thus $v \in [i, j]$, while $u \notin [i, j]$, and therefore $u \in O(i, j)$. We conclude that $Bag(n) \subseteq O(i, j)$.

Condition 2. Relations $O(i, k-1) \cap [k, j] = \varnothing$ and $[i, k-1] \cap O(k, j) = \varnothing$ can be shown using the same argument of Condition 1.

We now show $O(i, j) \cup \{v_k\} \subseteq Bag(n)$. Because $n = T(v_k)$, $v_k \in Bag(n)$ directly follows from Definition 4. Let $u \in O(i, j)$. Again, $u \notin [i, j]$ and there must exist $v \in [i, j]$ such that $u \in N(v)$. Since n is an anchored node, we could have $n = T(v)$ (which implies $v = v_k$) or else $T(v)$ is in one of the two subtrees rooted at the left child or at the right child of n, respectively. In both cases we have that $T(u)$ dominates n, and thus u appears in $Bag(n)$ by Definition 4. We therefore conclude that $O(i, j) \subseteq Bag(n)$.

In the other direction, let $u \in Bag(n)$. If $u = v_k$, we are done. Assume now that $u \neq v_k$. There must exist a vertex v such that $u \in N(v)$, with $T(v)$ in the subtree rooted at n and $T(u)$ dominating $T(v)$. Thus we have $v \in [i, j]$ and $u \notin [i, j]$, and therefore $u \in O(i, j)$. We conclude that $Bag(n) \subseteq O(i, j)$.

Condition 3. For an outside parse tree, we can consider both anchored unary nodes and anchored leaf nodes as special cases of an anchored binary node. Thus $Bag(n) = O(i, j) \cup \{v\}$ can be shown using an argument very similar to Condition 2.

We now prove the converse part of the theorem. We start by showing that a parse tree T meeting the empty intersection requirements in Conditions 1 and 2 is an outside parse tree. More precisely, we need to show that for every pair of vertices v, u with $u \in N(v)$, if $T(v)$ does not dominate $T(u)$ then $T(u)$ must dominate $T(v)$. Consider such a pair v and u. Suppose that, contrary to the theorem, $T(u)$ does not dominate $T(v)$. Let then n be the least common ancestor of $T(u)$ and $T(v)$. Because $T(u)$ and $T(v)$ are not in a dominance relation, n must be a binary node with children n_L and n_R: We consider the case in which $T(v)$ is in the subtree rooted at n_L, and $T(u)$ is in the subtree rooted at n_R (the other case is symmetrical). Let $[k, \ell]$ be the span of n, $[k, m]$ be the span of n_L and $[m', \ell]$ be the span of n_R, where $m' = m$ if n is an unanchored node, and $m' = m + 1$ if n is anchored. We then have $u \in O(k, m) \cap [m', \ell]$, which violates one of the empty intersection requirements in either Condition 1 or 2. We therefore conclude that $T(u)$ must dominate $T(v)$.

To complete the proof of the converse part of the theorem, we need to show that if Conditions 1 to 3 are met by the outside parse tree T, then T is an outside tree decomposition of G relative to $v_1 \cdots v_{|V|}$. But we have already shown that, for an outside parse tree, the collection of bags specified by Condition 2 in Definition 4 and the collection of bags specified by Conditions 1 to 3 are the same. ■

Similarly to the previous section, the importance of Theorem 5 is that we can reconstruct each set $Bag(n)$ only on the basis of its span and the spans of its children.

In this way, we can avoid storing the bags at T's nodes. In addition, we will use Theorem 5 to enforce on a generic parse tree the condition on branching structure for outside parse trees.

Algorithm 2 computes quantities $b(i, j)$ for each i, j with $0 \leq i < j \leq |V|$, with the same meaning as in Algorithm 1 but relative to the outside tree decomposition. The idea underlying Algorithm 2 is very similar to the one already discussed for Algorithm 1, and the correctness of Algorithm 2 directly follows from Theorem 5. To simplify the structure of the algorithm, we treat unary anchored nodes as binary anchored nodes for which one of the two child spans is empty. For this reason, for every i with $0 \leq i \leq |V|$, we define the null span $[i, i]$ to be the empty set and, accordingly, we let $O(i, i)$ be the empty set. We use this in line 12. Furthermore, we define $b(i, i)$ to be zero in line 4, such that this term does not affect the bag size of a unary node in the maximum operation of line 13.

Algorithm 2 Computation of the optimal width of G for outside tree decomposition.

```
1:  procedure OutsideTreeWidth(G = (V, E), v_1 ··· v_|V|)
2:      for each i do
3:          b(i − 1, i) ← |O(i − 1, i)| + 1                ▷ anchored leaf nodes
4:          b(i, i) ← 0                                    ▷ used for unary nodes in line 13
5:      for each i, j in ascending order of j − i ≥ 2 do
6:          b(i, j) ← +∞
7:          for each k with i < k < j do                   ▷ unanchored nodes
8:              if O(i, k) ∩ [k, j] = ∅ and [i, k] ∩ O(k, j) = ∅ then
9:                  b ← max{|O(i, j)|, b(i, k), b(k, j)}
10:                 b(i, j) ← min{b(i, j), b}
11:         for each k with i < k ≤ j do                   ▷ anchored nodes
12:             if O(i, k − 1) ∩ [k, j] = ∅ and [i, k − 1] ∩ O(k, j) = ∅ then
13:                 b = max{|O(i, j)| + 1, b(i, k − 1), b(k, j)}
14:                 b(i, j) ← min{b(i, j), b}
15:     return b(0, |V|) − 1
```

Theorem 6
Algorithm 2 finds the width of the optimal outside tree decomposition of the input graph relative to the input vertex order.

Proof. This can be shown using Theorem 5 and applying essentially the same argument developed in the proof of Theorem 2. ■

Theorem 7
An optimal outside tree decomposition of G relative to $v_1 \cdots v_{|V|}$ can be found in time $\mathcal{O}(|V|^3)$.

Proof. The loop structure of Algorithm 2 is that of standard CKY parsing, giving $\mathcal{O}(|V|^3)$ running time, if we assume that the intersection and cardinality operations on the sets

$O(i,j)$ can be performed in constant time. This is possible using some preprocessing, as we now explain in detail.

For each i,j with $0 \leq i < j \leq |V|$, we first define $N(i,j)$ to be a vector such that $N(i,j)[k]$, $1 \leq k \leq |V|$, is the number of vertices in the span $[i,j]$ that have vertex v_k as a neighbor:

$$N(i,j)[k] = |\{v \mid (v,v_k) \in E \text{ or } (v_k,v) \in E,\ v \in [i,j]\}|$$

These vectors can be computed by letting $N(i-1,i)[k] = 1$ if $(v_i,v_k) \in E$ or $(v_k,v_i) \in E$, and letting $N(i-1,i)[k] = 0$ otherwise. We then use the recurrence relation $N(i,j)[k] = N(i,j-1)[k] + N(j-1,j)[k]$. The entire computation can be carried out in time $\mathcal{O}(|V|^3)$.

Algorithm 2 needs the cardinality of sets $O(i,j)$, which can be computed from $N(i,j)$ in time $\mathcal{O}(|V|)$ for fixed i and j. The algorithm also requires intersecting $O(i,j)$ with spans that are either immediately adjacent to span $[i,j]$, or else separated from $[i,j]$ by a single vertex v_i or v_{j+1}. We precompute the highest index $l_0(i,j)$ less than or equal to i and the highest index $l_1(i,j)$ less than or equal to $i-1$ of a vertex in $O(i,j)$:

$$l_0(i,j) = \max\{k \mid k \leq i,\ N(i,j)[k] > 0\}$$
$$l_1(i,j) = \max\{k \mid k \leq i-1,\ N(i,j)[k] > 0\}$$

To determine whether an intersection such as $[i,k] \cap O(k,j)$ is empty, we simply compare $l_0(k,j)$ to i. Symmetrical quantities can be precomputed to determine whether $O(i,k) \cap [k,j]$ is empty. In summary, after $\mathcal{O}(|V|^3)$ preprocessing time, all set operations in Algorithm 2 can be performed in constant time, yielding overall running time of $\mathcal{O}(|V|^3)$.

Similarly to the proof of Theorem 3, we can augment Algorithm 2 by saving backpointers—that is, values of k at lines 10 and 14 that correspond to the computed minimum for each $b(i,j)$. Backpointers can later be used to retrieve optimal outside decompositions in time $\mathcal{O}(|V|)$. We can then annotate each node of the tree with the associated bag in time $\mathcal{O}(b(0,|V|))$ per node, using Theorem 5. ■

5. String-To-Graph Parsing

In this section we discuss the problem of string-to-graph parsing, as defined in Section 1, in the context of the formalism of HRGs. We briefly introduce HRG and then show how to extract HRGs from the tree decompositions developed in Sections 3 and 4. We also introduce a linguistically motivated model to score graphs, and develop a polynomial time (and space) algorithm for string-to-graph parsing using the extracted HRGs and the proposed model.

5.1 Hyperedge Replacement Grammars

HRGs are a general graph rewriting formalism (Drewes, Kreowski, and Habel 1997), which generalizes CFGs to generate graphs rather than strings. Formally, an HRG is defined by a tuple (N,Σ,R,S), where N is a ranked alphabet of hyperedge labels called nonterminals, Σ is an alphabet of edge labels called terminals with $N \cap \Sigma = \varnothing$, R is a set of rules, and $S \in N$ is a distinguished start symbol. Hyperedges are graph edges that may connect to an arbitrary number of vertices, rather than to two vertices as in the standard definition of an edge. The vertices that a nonterminal hyperedge connects to

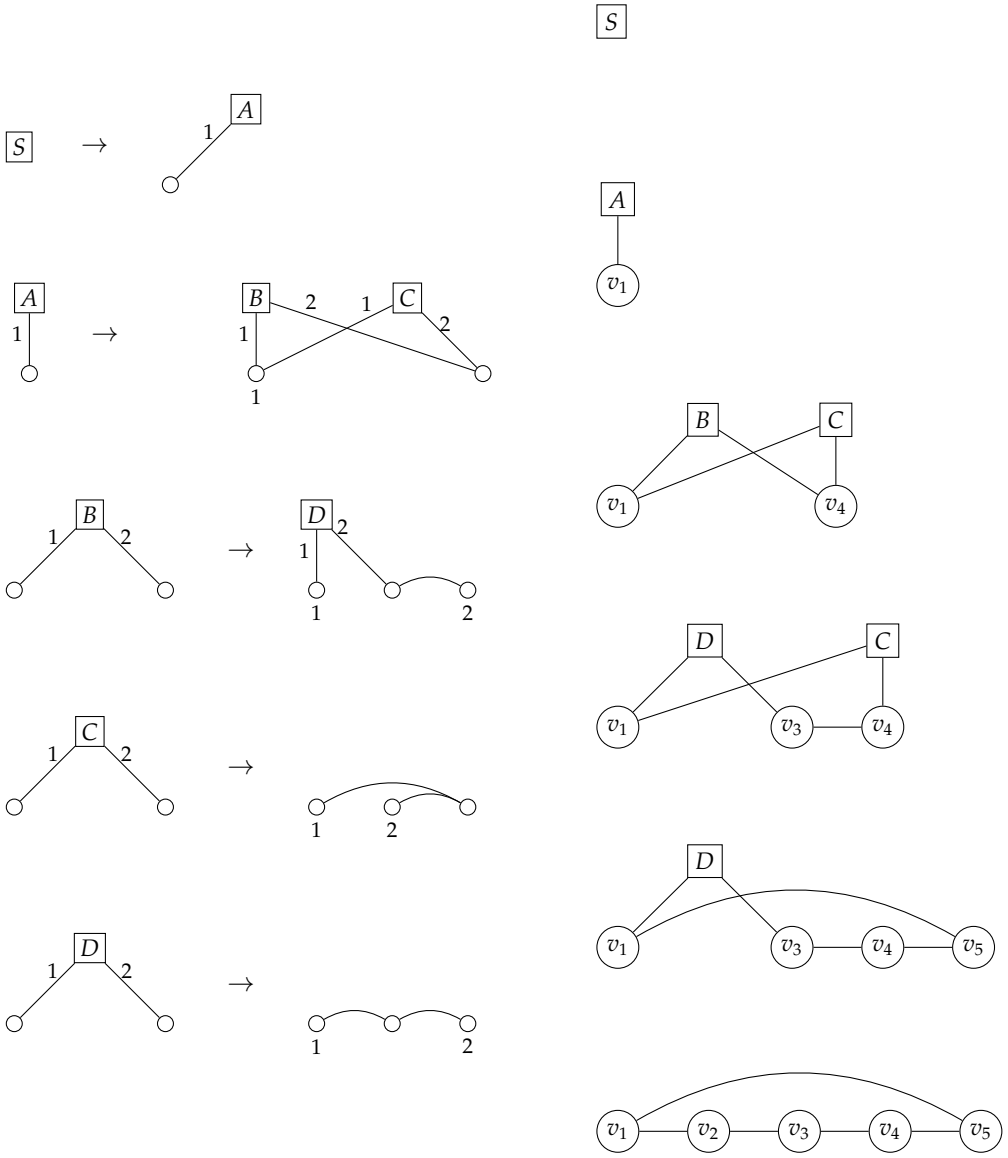


Figure 7
HRG rules (left column) extracted from the tree decomposition of Figure 6. The derivation (right column) of the graph of Figure 6 using the extracted grammar.

are known as its **attachment vertices**, and form an ordered set. A nonterminal symbol $A \in N$ of rank r has r attachment vertices. In the example HRG depicted in Figure 7, hyperedges are represented with squares, and the order of attachment vertices is indicated by the numbers adjacent to the lines connecting each hyperedge to its attachment vertices.

Each rule in R has the form $A \to h$, where $A \in N$, and h is a hypergraph. The rule indicates that we can replace any instance of A with a copy of h, consisting of new nonterminal hyperedges and new terminal edges. The attachment vertices of the

left-hand side nonterminal hyperedge A are also vertices of the right-hand side graph fragment h. To show this correspondence, in Figure 7, the numbers below vertices in the right-hand side graph fragment of each rule correspond to the attachment vertices of the left-hand side nonterminal hyperedge.

An HRG derivation starts with the distinguished start symbol S (which has zero attachment vertices), and proceeds by applying grammar rules to replace nonterminal hyperedges one at a time until no nonterminals are left. As with CFGs, a rule may apply whenever its left-hand side nonterminal is seen, regardless of any other features of the derivation.

Example 8
In the grammar in the left column of Figure 7, we have a rule with left-hand side hyperedge B with two attachment vertices. Attachment vertices are numbered with integers 1 and 2 at the lines connecting the hyperedge to its attachment vertices, and integers 1 and 2 also label the vertices at which the graph fragment in the right-hand side is attached to the rest of the graph. This rule rewrites B into a graph fragment consisting of a new nonterminal hyperedge D and a terminal edge. In the derivation in the right column of Figure 7, a hyperedge labeled B impinges upon vertices v_1 and v_4. When B is rewritten by this rule, the right-hand side graph of that rule is attached to the graph being derived, at vertices v_1 and v_4, which correspond to the attachment vertices. As a result, a new hyperedge D is introduced incident upon v_1 and v_3, and a new terminal edge is introduced incident upon v_3 and v_4.

Because rewriting is carried out independently of the derivation context, an HRG derivation of a graph G can be viewed as a derivation tree T. This tree has a grammar rule at each node, where the rule expanding a nonterminal hyperedge is a child of the rule that introduced the nonterminal in its right-hand side. The derivation tree provides a tree decomposition of the derived graph. More precisely, the tree decomposition has the same nodes and the same branching structure as the derivation tree, and each bag in the tree decomposition contains all the vertices that appear in the right-hand side of the rule. The derivation tree must satisfy the three properties in the definition of a tree decomposition:

1. Vertex cover: Each vertex in G must be generated by some rule in the HRG derivation tree T.
2. Edge cover: Each edge (u, v) in G must also be generated by some rule r, and therefore both u and v appear in the right-hand side of r, and in the node for r in T.
3. Running intersection: Any vertex v of G is first generated by some rule r of T. Vertex v can only appear in another rule r' if it is connected to the lefthand side nonterminal of r', and therefore appears in the parent of r' in T. Thus, all occurrences of v form a connected subtree of T.

See also Lautemann (1988) for further discussion.

5.2 HRG Extraction

We have shown that a derivation of a graph by an HRG can be viewed as a tree decomposition of the graph itself. Conversely, it is also possible to extract HRG rules from a tree decomposition of a graph. In this process, which we describe precisely next,

each node n in the tree decomposition becomes a rule r_n in the extracted grammar, and the arcs from node n to its children are the nonterminals in the righthand side of the rule r_n. An example of an extracted grammar is shown in Figure 7 and is discussed in Example 9. This grammar was extracted from the outside tree decomposition of Figure 6.

The process of grammar extraction from a tree decomposition begins by assigning each edge in the graph to a node in the tree decomposition: The definition of tree decomposition (Definition 1) ensures that this is always possible. We next construct the nonterminals and the rules of the grammar.

- We extract the set of nonterminal symbols of the grammar from the arcs in the tree decomposition. In Figure 7, nonterminal symbols A, B, C, and D were assigned arbitrarily; other schemes for choosing nonterminal symbols in the extracted grammar are also possible, as will be discussed in the next subsection. For an arc (n, n') of the tree decomposition, consider the associated bags $Bag(n)$ and $Bag(n')$, respectively. The set of vertices $Bag(n) \cap Bag(n')$, known as the arc's **separator**, are the attachment vertices of the corresponding HRG nonterminal, that is, the vertices to which the nonterminal was connected before being rewritten.
- We extract the set of rules of the grammar from the nodes in the tree decomposition. The rule at a given node rewrites the nonterminal at the node's incoming arc to the nonterminals at the node's outgoing arcs, and also produces the edges of the original graph that are assigned to this bag. All of the vertices in the bag are vertices of the (hyper)graph fragment in the right-hand side of the extracted rule. This includes vertices that are attachment vertices of the nonterminal hyperedges in the left- and right-hand side of the rule, as well as vertices not connected to any hyperedge, that is, vertices that appear in this bag but are not part of the neighbor bags in the tree decomposition.

Example 9
Coming back to our running example, consider the outside tree decomposition of Figure 6 and the node n with span $[1, 5]$ and $Bag(n) = \{v_1, v_4\}$. We extract from n the HRG rule in the left column of Figure 7 that rewrites hyperedge A into hyperedges B and C. Note that A has a single attachment vertex, obtained as the intersection of $Bag(n)$ and the bag at n's parent. Both B and C have two attachment vertices, obtained as the intersection of $Bag(n)$ and the bag at n's left and right child, respectively.

As an example of a leaf node in the tree decomposition, consider node n' with span $[4, 5]$ and $Bag(n') = \{v_1, v_4, v_5\}$. We extract from n' the HRG rule that rewrites hyperedge C. Because this bag is a leaf of the tree decomposition, the corresponding HRG rule has no right-hand side nonterminal hyperedges. The nonterminal hyperedge C has two attachment vertices, corresponding to the intersection of the bags at n' and its parent. The rule produces one additional vertex, corresponding to the vertex v_4, which is in $Bag(n')$ and not in the bag at its parent.

This process of rule extraction can be applied to any tree decomposition, even when the vertices of the graph are unordered. For the task of string-to-graph parsing, in which the input string represents an ordered sequence of vertices, we augment the rule extraction process by assigning an order to the vertices within each rule—this is the

order in which the vertices of the bag appear in the original graph, and is shown in Figure 7 by the left to right position of vertices in each rule's right-hand side.

We further augment each rule by designating which of the rule's vertices are **anchored**, if any. Each vertex in the original graph must appear as an anchored vertex in exactly one of the extracted rules. When extracting HRG rules from an inside tree decomposition, only the leaf nodes contain anchored vertices, in particular the single vertex in the leaf's span. When extracting HRG rules from an outside tree decomposition, the anchored vertices of a rule consist of the anchored vertices at the corresponding node in the outside parse tree defined in Section 4. For both types of tree decomposition, there are either zero or one anchored vertices in each rule.

5.3 Scoring Graphs

We now develop an algorithm for finding the best scoring graph under a general framework that allows the score of each rule in the HRG derivation to depend on the positions of the rule vertices in the input string. Mathematically, we require that the score of the output graph must decompose over the rules in the HRG derivation:

$$\text{score}(G) = \prod_r \text{score}(r, \phi) \tag{1}$$

where the score of each rule is a function of the mapping ϕ from the rule's vertices to positions in the input string, and the score function may depend on any features of the input string, which is fixed during parsing. The function ϕ performs a role similar to lexicalization in parsing with CFGs. For example, consider a rule r with two vertices in its right-hand side, producing a terminal edge with a label "ARG1" from vertex 1 to vertex 2. Then $\text{score}(r, \{1 \rightarrow 6, 2 \rightarrow 7\})$ may be high if vertex 6 of the input sentence is the word "drink" and vertex 7 is the word "water." In general, the function ϕ provides a form of lexicalization parameterized by not just one or two words as is generally the case with lexicalized CFG parsing, but rather parameterized by as many words as there are vertices in the right-hand side of the HRG rule.

5.3.1 Valence/Relation Model. As a concrete, linguistically motivated example of the general model in Equation (1) for scoring graphs, we define a specific model for scoring graphs in terms of individual relations they contain. This model is based on the idea that each word in an input sentence has a valence, specifying the set of relations in which it participates, and that each relation selects for vocabulary items in its two argument positions. We define this model to provide an example of how the general form of Equation (1) relates to specific features that one would want in a system for semantic parsing. Equation (1) is very general and subsumes many models more complex than the valence/relation model.

The valence/relation model is similar to many models used in practice in semantic and syntactic parsing, including the DMV model for unsupervised parsing of Klein and Manning (2004) and the semantic parsing model of Toutanova, Haghighi, and Manning (2005), among many others. So-called arc-factored models in dependency parsing correspond to the relation model without valence, and have been found not to achieve state-of-the-art results. The best parse of a sentence under the valence/relation model can be found with the algorithm that will be presented in Section 5.4. The best parses for either the valence model or the relation model on their own can be found more efficiently. The joint valence/relation model is the simplest model we have found

that illustrates the power of the general HRG parsing algorithm and the importance of finding rules with low treewidth.

We assume that the score of a semantic graph can be decomposed into two types of terms:

$$\text{score}(G) = \text{val}(G)\text{rel}(G) \tag{2}$$

The first term in the right-hand side of (2), which we call the **valence** term, assigns a score to each word in the input sentence as a function of the set of relations that the word participates in, or, in terms of the graph, the multiset of labels of the edges of the graph having a vertex v_i as an endpoint:

$$\begin{aligned}
\text{val}(G) &= \prod_{v_i \in V} \text{val}(v_i, \text{out}(v_i), \text{in}(v_i)) \\
\text{in}(v_i) &= \left\{ \ell \mid (v_j, v_i, \ell) \in E \right\} \\
\text{out}(v_i) &= \left\{ \ell \mid (v_i, v_j, \ell) \in E \right\}
\end{aligned}$$

where $\text{in}(v_i)$ and $\text{out}(v_i)$ are multisets, that is, they also account for the multiplicity of their elements.

The second term in Equation (2), which we call the **relation** term, assigns a score to each relation as a function of the two words involved. In terms of the graph, this translates into a score for each edge, which is defined as a function of the edge label and vertices at the two endpoints of the edge:

$$\text{rel}(G) = \prod_{(v_i, v_j, \ell) \in E} \text{rel}(v_i, v_j, \ell)$$

In the extracted HRG, the valence of a vertex is determined by the rule at which the vertex is anchored. In addition, grammar nonterminals are used along the derivation to enforce the valence constraints at each of their attachment vertices, by tracking how many of the edge labels remain to be generated. Informally, for each of its attachment vertices, a nonterminal encodes a multiset of edge labels. When the nonterminal takes part in a derivation, the multiset records how many of the edge labels specified by the valence constraint at the attachment vertex remain to be generated. More precisely, consider a rule r of an extracted HRG, and let v be the vertex anchored at r. With an HRG extracted from an outside tree decomposition, all edges incident upon v are produced either at r itself, or else at some rule below r in the derivation. A grammar nonterminal therefore indicates which edge labels must be generated below that nonterminal in the derivation. With an HRG extracted from an inside tree decomposition, all edges incident upon v are produced at some rule above r in the derivation. A grammar nonterminal therefore tracks which edge labels have been already generated above that nonterminal in the derivation. To generalize to other possible types of tree decomposition, we can indicate, for each edge in a nonterminal's description, whether it is to be produced inside the subtree rooted at the nonterminal, or else whether it has already been produced outside the subtree rooted at the nonterminal.

Example 10
Let us consider an outside tree decomposition of a graph. We demonstrate how to extract an HRG that generates the graph according to the valence/relation model. Figure 9 shows an example grammar along with a derivation. The grammar has been extracted from an outside tree decomposition of the edge labeled graph in Figure 8, where the tree decomposition has the same shape as the one in Figure 6.

The grammar nonterminals in Figure 9 specify, for each of their attachment vertices, the multiset of edge labels to be produced on outgoing edges, and the multiset of edge labels to be produced on incoming edges. As already mentioned, these edge labels must be generated below the nonterminal, since the grammar has been extracted from an outside tree decomposition. For instance, the binary rule r in Figure 9 expands a nonterminal symbol $(1 : \{a, e\}, \{\})$. This symbol states that at attachment vertex 1 two outgoing edges are still missing, with labels a and e, while no incoming edge needs to be generated for this vertex. Rule r produces nonterminals $(1 : \{a\}, \{\}; 2 : \{\}, \{c\})$ and $(1 : \{e\}, \{\}; 2 : \{d\}, \{\})$. From these nonterminals we see that the generation of label a is dispatched to the first nonterminal and the generation of label e is dispatched to the second nonterminal. Furthermore, a new vertex v is introduced by r, connected to attachment vertex 2 of both the first and the second nonterminals in r's right-hand side. According to the tree decomposition in Figure 6, vertex v is anchored at r and the valence constraint for v is determined at r itself. Specifically, v needs an outgoing edge with label d and an incoming edge with label c, which are dispatched to the first and to the second nonterminals, respectively, in r's right-hand side. These two edge labels are later generated by the two unary rules which expand the two nonterminals in r's right-hand side.

For a rule r, we define edges(r) as the set of (terminal) edges in the rule's right-hand side. We also define anchored(r) to be a set of triples $(i, \text{out}, \text{in})$, where i is a vertex anchored in the rule r, out is the multiset of labels of i's outgoing edges in the final graph to be generated, and in is the multiset of labels of i's incoming edges. In the specific case of HRGs extracted from inside or outside tree decompositions, anchored(r) contains at most one element; for more general cases, this set might contain more than one element. The valence/relation model can be put into the general form in Equation (1) by assigning the score of each edge to the rule that produces it, and the valence score of each vertex to the unique rule at which the vertex itself is anchored:

$$\text{score}(r, \phi) = \prod_{(i,\text{out},\text{in}) \in \text{anchored}(r)} \text{val}(\phi(i), \text{out}, \text{in}) \prod_{(i,j,\ell) \in \text{edges}(r)} \text{rel}(\phi(i), \phi(j), \ell)$$

5.4 Parsing Algorithm

We show in this section one of the main results of this article, namely, that string-to-graph parsing can be carried out in polynomial time (and space) for HRGs extracted

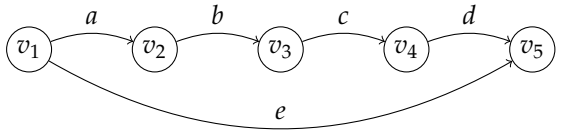


Figure 8
A graph with labeled edges.

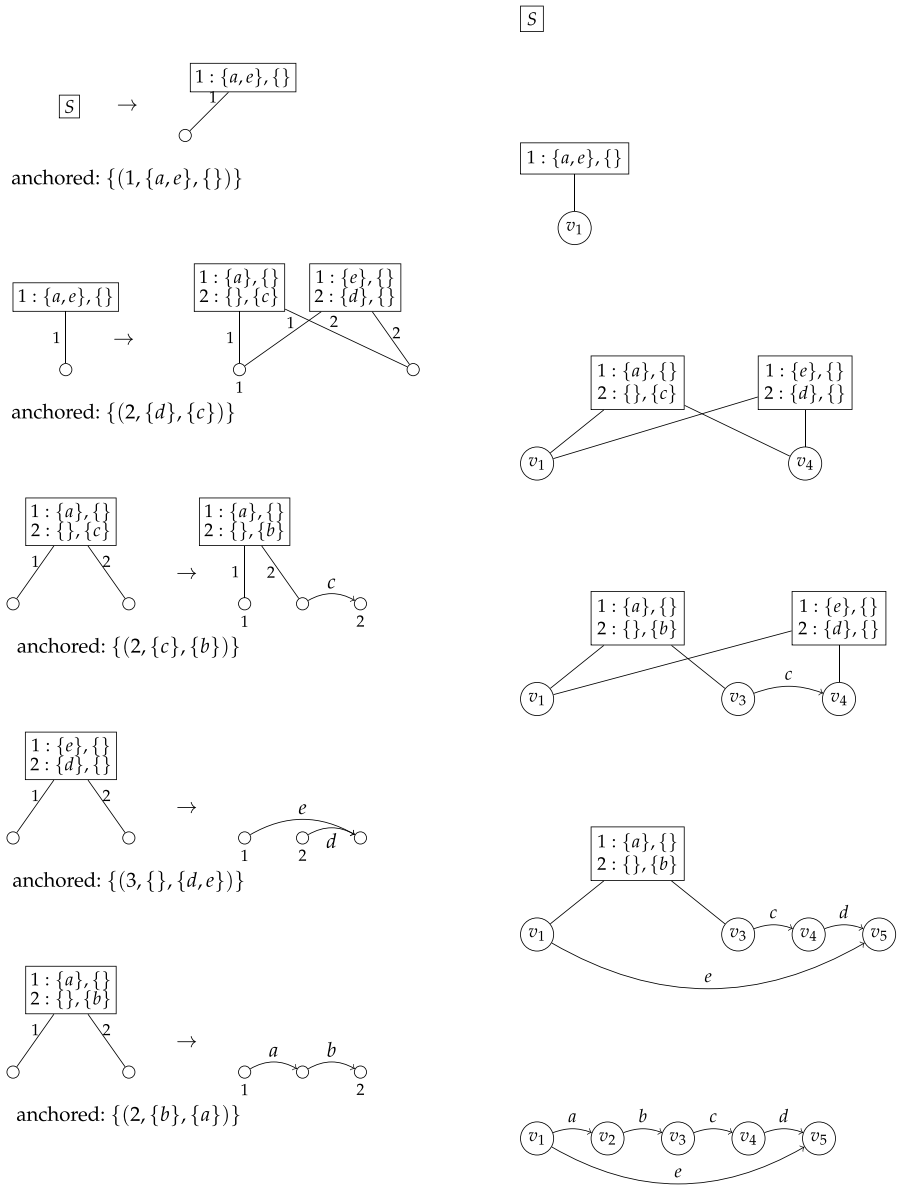


Figure 9
HRG rules (left column) extracted from the graph G of Figure 8, and a derivation (right column). Grammar nonterminals track which edges that were required by the valence remain to be produced.

from inside or outside tree decompositions of bounded treewidth. We do so by presenting a dynamic programming algorithm for string-to-graph parsing, formulated in an abstract form that works for both inside and outside HRGs. The algorithm takes as input a string representing an ordered sequence of vertices, and an HRG augmented as described in Section 5.3.1, and it finds the graph over the input string with the highest score according to the scoring function in Equation (1).

We need to introduce some additional notation. We schematically write $A \to BC$ and $A \to \varnothing$ to denote HRG rules having two and zero nonterminal hyperedges, respectively,

in their right-hand side. In order to simplify the presentation of the algorithm, we omit the unary rules of outside HRGs: One can easily process those rules by treating them as a special case of the anchored binary rules. Following Section 5.3, for an HRG rule r we write ϕ_r to denote a mapping of the vertices in the right-hand side of r to positions in the input string. If A is a hyperedge nonterminal appearing in the left-hand side or in the right-hand side of r, we also write $\phi_{r|A}$ to denote the restriction of ϕ_r to the attachment vertices of A.

The pseudocode for our method is reported in Algorithm 3. The algorithm is very similar to the CKY method for parsing based on context-free grammars in Chomsky normal form. Given as input a string $v_1 \cdots v_{|V|}$, the algorithm uses a $(|V|+1) \times (|V|+1)$ array δ to store the score of optimal subderivations. More precisely, consider a substring $s = v_{i+1} \cdots v_j$ of the input along with some mapping $\phi_{r|A}$ of the attachment vertices of nonterminal A to positions in the input string. Then we store in $\delta(A, i, j, \phi_{r|A})$ the score of an optimal derivation by the grammar of a graph fragment spanning s and with attachment vertices provided by $\phi_{r|A}$. In this way, if S is the start symbol of the grammar, $\delta(S, 0, |V|, \varnothing)$ provides the maximum score of any graph over the input string.

Algorithm 3 HRG parsing with edge weights.

```
1:  procedure HRGPARSE(v_1 ··· v_|V|, HRG rules)
2:      for each i, j do                                          ▷ initialize chart
3:          for each r, φ_r|A with A the left-hand side of r do
4:              δ(A, i, j, φ_r|A) ← −∞
5:      for each i do                                             ▷ terminal rules
6:          for each r : A → ∅ do
7:              for each φ_r with anchored vertex fixed to v_i do
8:                  δ(A, i − 1, i, φ_r|A) ← score(r, φ_r)
9:      for each i, j in ascending order of j − i > 1 do          ▷ binary rules
10:         for each k with i < k < j do
11:             for each rule r : A → BC do
12:                 for each φ_r with anchored vertex of r (if any) fixed to v_k do
13:                     b ← δ(B, i, k, φ_r|B) δ(C, k + α(r), j, φ_r|C) score(r, φ_r)
14:                     δ(A, i, j, φ_r|A) ← max{δ(A, i, j, φ_r|A), b}
15:     return δ(S, 0, |V|, ∅)
```

The main difference with respect to the CKY algorithm is that, in addition to the computation of the rule endpoints i, j, and k, we must also specify the location in the input string of the endpoints of any other vertices used in the current rule, in order to evaluate the rule's score, and to ensure consistency with other rules in the derivation. This necessitates an additional loop over all possible choices for the mapping ϕ_r, which maps vertices of the HRG rule to positions in the input string. In general, the range of the function ϕ_r can contain either vertices inside the span $[i, j]$ of the current CKY combination step, or vertices outside the span $[i, j]$. When using a grammar extracted from inside tree decompositions, ϕ_r will only refer to vertices inside $[i, j]$, and when using a grammar extracted from outside tree decomposition, ϕ will only refer to vertices outside $[i, j]$.

Each terminal rule $r : A \to \varnothing$ is processed by the nested loops at line 5 by anchoring it to some vertex v_i. In case of inside HRGs, v_i is the only attachment vertex for r's right-hand side. In case of outside HRGs, the attachment vertices are the neighbors of

v_i, which are selected through the choice of mapping $\phi_{r|A}$ and then used to compute the score for the generated graph fragment.

In order to process binary rules $r : A \rightarrow BC$, we need to account for the spans of the nonterminals B and C, which depend on whether r is anchored or not. Specifically, if r is anchored we need to introduce a vertex in between B and C. We use a special term $\alpha(r) = 1$ for anchored r and $\alpha(r) = 0$ for unanchored r. Rule r is then processed by the nested loops at line 9, which use index k and $\alpha(r)$ to indicate the boundary between the spans of B and C. We choose a mapping ϕ_r from r's vertices to the input vertices, with the proviso that if r is anchored then r's anchored vertex is mapped to v_k. We then select optimal subgraphs derived by B and C that are compatible with ϕ_r, compute the score of a new subgraph derived from A via r at line 13, and update table δ at line 14.

The running time of Algorithm 3 is $\mathcal{O}(|G||V|^{3+d_r})$. Here $|G|$ is the number of rules in the input HRG, $|V|$ is the length of the input string, and d_r is the maximum number of unanchored vertices in an HRG rule, since anchored vertices have their positions already fixed by the indices of the rule boundaries i, k, and j. We observe that d_r is at most one plus the treewidth of the tree decomposition from which the grammar was extracted, because we use at most one anchored vertex per rule. Both Algorithm 1 and Algorithm 2 can be easily extended to minimize the number of *unanchored* vertices per rule. For simplicity, the versions presented in this article minimize the *total* number of vertices per rule. Thus, we conclude that string-to-graph parsing can be carried out in polynomial time for HRGs extracted from inside or outside tree decompositions of bounded treewidth. Furthermore, finding low treewidth decompositions during grammar extraction leads to efficient parsing. Both the inside and outside algorithms for finding tree decompositions provide heuristics for efficiently finding tree decompositions of low treewidth.

6. Experiments

We evaluate our tree decomposition algorithms on Abstract Meaning Representation (AMR) (Banarescu et al. 2013). Figure 10 shows an example of an AMR in which the nodes represent the AMR concepts, and the edges represent the relations between the

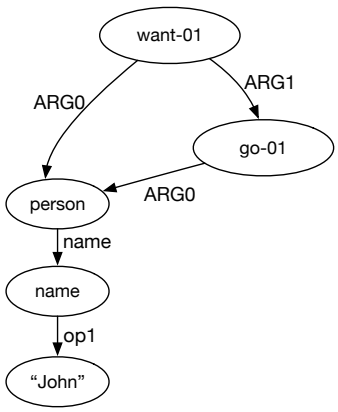


Figure 10
An example of AMR graph representing the meaning of "John wants to go."

Table 1
Rule and width statistics for the SemEval 2016 data set using inside and outside algorithms.

algorithm	average treewidth	rule types (labeled)	rule types (direction only)	rule types (unlabeled)
inside	1.97	20,772	5,788	3,902
outside	1.81	21,147	5,745	4,108

concepts. We can also see the reentrancy to the concept "person" for the control verb structure "want to go," where "want-01" and "go-01" have the same subject. AMR has been used in various applications such as text summarization (Liu et al. 2015), sentence compression (Takase et al. 2016), and event extraction (Huang et al. 2016). We use the training set of LDC2015E86 for SemEval 2016 task 8 on meaning representation parsing (May 2016), which contains 16,833 pairs of sentence and AMR graph. This data set covers various domains including newswire and Web discussion forums.

We generate the vertex order for each AMR graph using the automatically generated alignments from an aligner by Pourdamghani et al. (2014), which aligns sentence tokens to concepts or relations in the graph. We collapse subgraphs of named entities, dates, and verbalization[2] to a single node of the graph. For example, in Figure 10 the subgraph corresponding to the word "John" is collapsed to a single node "person-John." AMR also uses predicate–argument structure extensively: for example, "teacher" is verbalized as "(person: ARG0-of teach-01)." We represent this verbalization as a single node "person-ARG0of-teach-01." We want to generate the vertex order of the graph in a way that preserves as much as possible the order of the sentence tokens. To this end, we sort all aligned vertices of the AMR graph according to the position of the associated token in the input sentence, and insert the unaligned vertices according to their order in the depth-first traversal of the graph.

Some error is introduced by the use of automatically generated alignments; unfortunately, we have no gold alignments against which to measure the automatic aligner. Anecdotally, aligner errors seem to increase treewidth, because alignment errors create spurious cases of complex reordering between the AMR and the string.

After we have generated the vertex order for each AMR graph, we run Algorithm 1 and Algorithm 2 to compute the optimal inside and outside statistics, reported in Table 1. We can see that the average treewidth for both algorithms is smaller than 2: Such small values provide efficient time bounds for the string-to-graph parsing algorithm of Section 5.4. Furthermore, outside tree decompositions have treewidth slightly smaller than inside tree decompositions on average. The full distribution of widths is shown in Figure 11. (Treewidth of zero results from graphs with one vertex and no edges.) Allowing a maximum width of 5 covers 99% of sentences using either inside tree decompositions or outside tree decompositions. The maximum width is 14 for the inside algorithm and 11 for the outside.

As a term of comparison, the cache transition-based system for string-to-graph parsing presented by Gildea, Satta, and Peng (2018) also entails a tree decomposition of the parsed graph. We report here that the average treewidth from the cache transition system is 2.8, a value considerably larger than those in Table 1, in relative terms. This is

2 `http://amr.isi.edu/download/lists/verbalization-list-v2.06.txt`.

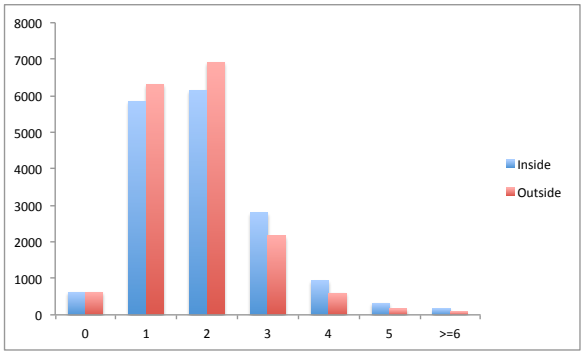


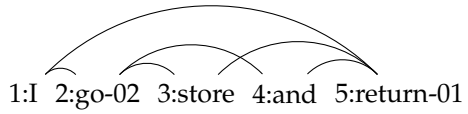
Figure 11
Distribution of treewidths of AMR graphs.

because the cache transition-based system processes vertices strictly from left to right, and the associated tree decomposition has a left to right bias. In contrast, the inside and outside algorithms for optimizing tree decompositions use more flexible bi-directional strategies in their search.

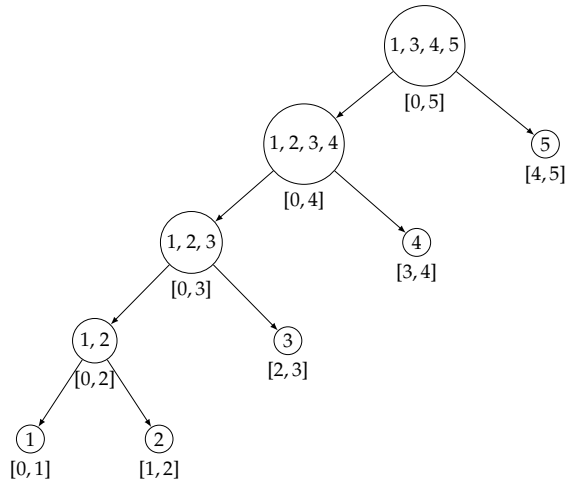
We also extract HRG rules from the optimal tree decompositions of Algorithm 1 and Algorithm 2, using a single HRG nonterminal for each possible number of attachment vertices, and count the number of distinct rules generated. The edges in the AMR graph are labeled and directed. If we keep the labels and directions of the edges, outside tree decomposition produces a larger number of rule types than inside tree decomposition, even though the treewidth is smaller. This might be because the rules of the outside decomposition usually have additional labeled outgoing edges that connect to the attachment vertices. If we ignore the edge labels and only account for the directions, the outside generates slightly fewer rule types than the inside tree decomposition. If we ignore both the labels and the directions of the edges, inside tree decomposition produces a smaller number of rule types.

In Figure 12 we show the AMR graph from our data set for the sentence "I went to the store and back." For this graph outside treewidth is 2 and inside treewidth is 3. The problem here is that we cannot find an inside decomposition where each span has a total number of connections to vertices outside the span itself (the attachment vertices) that is smaller than 3. For instance, if we parse the sentence into spans $[0,4]$ and $[4,5]$, we have $I(0,4) = \{1,3,4\}$ and $I(4,5) = \{5\}$, which adds up to a bag of 4 vertices at the node with span $[0,5]$. However, we also observe that while the span $[0,4]$ has three connections, to outside vertices, all of the connections are made to the same vertex 5. In the outside decomposition, the outside connections of span $[0,4]$ would not be repeatedly counted and the span would have a contribution of 1 to the total width, resulting in a decomposition with smaller width.

In contrast, an inside tree decomposition can also produce a treewidth smaller than an outside tree decomposition. In Figure 13 we show the AMR graph from our data set for the sentence "How can a country stake its claim to its right?". Whereas the reported inside decomposition has width 2, there is no outside decomposition that has width lower than 3. This is because when we introduce vertex 3 from left to right, it connects to every vertex on its right, resulting in a width of 3. This also goes to vertex 4 when



Inside tree decomposition:



Outside tree decomposition:

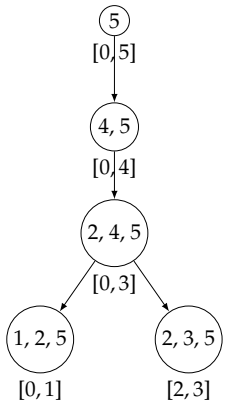


Figure 12
AMR graph for the sentence "I went to the store and back" (edge directions omitted) along with inside and outside tree decompositions. The outside decomposition has smaller width in this case.

we introduce it from right to left, because vertex 4 has connections to every vertex to its left. Finally, we could parse the graph bidirectionally, introducing vertices 3 and 4 at the bottom of our parse tree and then proceeding in an outward fashion, but we will face the same problem since vertices 3 and 4 have connections to each other vertex outside the span $[2, 4]$. We then see that one factor that contributes to higher outside width is when a single vertex in a span has many connections to vertices outside of the span itself. In a sense, this problem represents the symmetrical scenario with respect to the problem discussed for the example in Figure 12. What we learn then from our statistics in Table 1 is that, for the data set at hand, the first scenario is more frequent than the second one.

We also experiment on four data sets from the 2007 CoNLL Shared Task on dependency parsing: English, Czech, Italian, and Arabic. For these data sets, the structures at hand are always trees, and might be projective as well as non-projective. Table 2 shows the width statistics for these four languages. We observe that the outside tree decomposition consistently produces smaller width than the inside tree decomposition on average. As for the extracted HRG rules, we only consider labeled rules here. We can see that the number of rule types using inside tree decomposition is lower than outside tree decomposition.

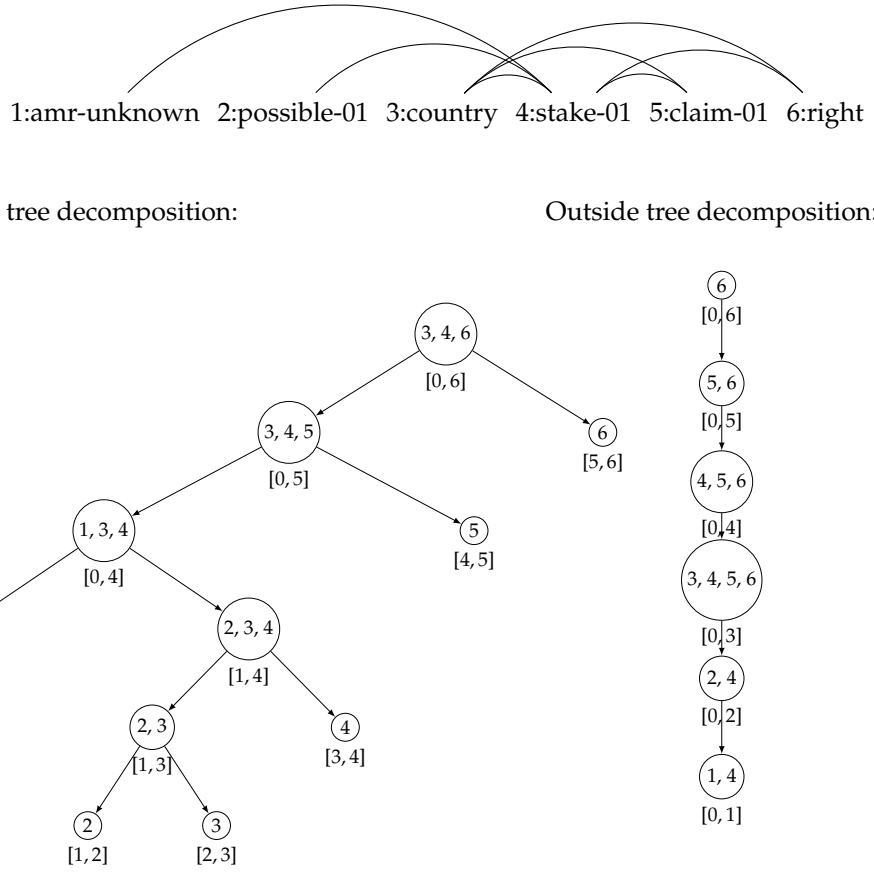


Figure 13
AMR graph for the sentence "How can a country stake its claim to its right?" (edge directions omitted) along with inside and outside tree decompositions. The inside decomposition has smaller width in this case.

Table 2
Rule and width statistics for the 2007 CoNLL Shared Task on dependency parsing.

Language	# sentences	width (inside)	width (outside)	rule types (inside)	rule types (outside)
English	18,577	1.07	1.06	734	1,137
Czech	25,364	1.15	1.07	2,736	3,429
Italian	3,110	1.05	1.02	348	475
Arabic	2,912	1.09	1.03	562	868

7. Discussion

Gildea, Satta, and Peng (2018) define a cache transition system for string-to-graph parsing, and Peng et al. (2018) apply the system to the task of AMR parsing. This system is also based on tree decompositions of the graphs being produced, where the structure

of the tree decomposition can be read off of the sequence of push and pop operations of the transition system. This family of tree decompositions is called here **cache tree decomposition**. It is possible to show that, if we restrict the anchored rules of our outside tree decomposition to rules anchored at the leftmost vertex in the span, we obtain a binary version of the cache tree decomposition that preserves its width. This means that outside grammars implement a generalization of the parsing strategy exploited by the cache transition system. The algorithms presented in this article often result in simpler analyses of the graphs than those provided by cache tree decompositions, as evidenced by lower treewidth. One area for future research is the design of parsers that combine the advantages of linear-time left-to-right processing from the cache transition system and lower complexity of states and graph-building operations from the algorithms in this article.

We have pointed out that inside and outside decompositions are complementary to each other, in the sense that in an inside decomposition the attachment vertices of a span are those vertices inside the span that connect to outside vertices, while, in an outside decomposition, the attachment vertices of a span are those vertices outside the span that connect to inside vertices. In Section 6 we have also discussed specific graphs that challenge either the first or else the second features above. In this respect, it would be interesting to investigate more sophisticated strategies for tree decomposition that mix the two features above, in order to achieve an even smaller treewidth on average. As a general problem definition for these strategies, we wish to find the decomposition of the input graph with minimal width, such that the vertices appear as anchored vertices at leaves of the tree decomposition in the input vertex order. (Although the outside decomposition has anchored vertices in internal nodes of the tree, this can be converted to a normal form with anchored vertices only at leaves.) It is an open question whether a polynomial-time algorithm can be found for this problem, or, alternatively, whether it can be shown to be NP-hard.

Acknowledgments

This research was supported by NSF awards 1449828 and 1813823. We are grateful to three anonymous reviewers and Parker Riley for their comments.

References

Aalbersberg, I. J. J., G. Rozenberg, and A. Ehrenfeucht. 1986. On the membership problem for regular DNLC grammars. *Discrete Applied Mathematics*, 13(1):79–85.

Aho, Albert V. and Jeffery D. Ullman. 1972. *The Theory of Parsing, Translation, and Compiling*, volume 1. Prentice-Hall, Englewood Cliffs, NJ.

Banarescu, Laura, 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 and Interoperability with Discourse*, pages 178–186, Sofia.

Björklund, Henrik, Frank Drewes, and Petter Ericson. 2016. Between a rock and a hard place—uniform parsing for hyperedge replacement DAG grammars. In A.-H. Dediu, J. Janoušek, C. Martín-Vide, and B. Truthe, editors, *Language and Automata Theory and Applications*, pages 521–532, Springer International Publishing, Prague.

Chiang, David, 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 (Volume 1: Long Papers)*, pages 924–932, Sofia.

Collins, Michael John. 1996. A new statistical parser based on bigram lexical dependencies. In *Proceedings of the 34th Annual Conference of the Association for Computational Linguistics (ACL-96)*, pages 184–191, Santa Cruz, California.

Damonte, Marco, B. Shay Cohen, and Giorgio Satta. 2017. An incremental parser

for abstract meaning representation. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 1, Long Papers*, pages 536–546, Valencia, Spain.

Dohare, Shibhansh and Harish Karnick. 2017. Text summarization using abstract meaning representation. *CoRR*, abs/1706.01678.

Drewes, Frank, Berthold Hoffmann, and Mark Minas. 2015. Predictive top–down parsing for hyperedge replacement grammars, In F. Parisi-Presicce and B. Westfechtel, editors, *Graph Transformation*, pages 19–34, Springer International Publishing.

Drewes, Frank, Hans-Jörg Kreowski, and Annegret Habel. 1997. Hyperedge replacement graph grammars. In G. Rozenberg, editor, *Handbook of Graph Grammars*, volume 1, World Scientific, Singapore, pages 95–162.

Eisner, Jason M. 1996. Three new probabilistic models for dependency parsing: An exploration. *In Proceedings of the 16th Conference on Computational Linguistics (COLING-96)*, pages 340–345, Copenhagen.

Flanigan, Jeffrey, Sam Thomson, Jaime 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 (ACL-14)*, pages 1426–1436, Baltimore, MD.

Gildea, Daniel, Giorgio Satta, and Xiaochang Peng. 2018. Cache transition systems for graph parsing. *Computational Linguistics*, 44(1):85–118.

Graham, S. L. and M. A. Harrison. 1976. Parsing of general context free languages. In M. Rubinoff and M. C. Yovits, editors, *Advances in Computers*, volume 14. Academic Press, New York, NY, pages 77–185.

Huang, Liang and Kenji Sagae. 2010. Dynamic programming for linear-time incremental parsing. In *Proceedings of the 48th Annual Meeting of the Association for Computational Lignuistics (ACL-10)*, pages 1077–1086, Uppsala.

Huang, Lifu, Taylor Cassidy, Xiaocheng Feng, Heng Ji, Clare R. Voss, Jiawei Han, and Avirup Sil. 2016. Liberal event extraction and event schema induction. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL-16)*, volume 1, pages 258–268, Berlin.

Jones, Bevan, Jacob Andreas, Daniel Bauer, Karl Moritz Hermann, and Kevin Knight. 2012. Semantics-based machine translation with hyperedge replacement grammars. In *Proceedings of the 24th International Conference on Computational Linguistics (COLING-12)*, pages 1359–1376, Mumbai.

Jones, Bevan K., Sharon Goldwater, and Mark Johnson. 2013. Modeling graph languages with grammars extracted via tree decompositions. In *Proceedings of the 11th International Conference on Finite-State Methods and Natural Language Processing (FSMNLP2013)*, pages 54–62, St. Andrews.

Khashabi, Daniel, Tushar Khot, Ashish Sabharwal, and Dan Roth. 2018. Question answering as global reasoning over semantic abstractions. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-18)*, pages 1905–1914, New Orleans, LA.

Klein, Dan and Christopher D. Manning. 2004. Corpus-based induction of syntactic structure: Models of dependency and constituency. In *Proceedings of the 42nd Annual Conference of the Association for Computational Linguistics (ACL-04)*, pages 478–485, Barcelona.

Kübler, Sandra, Ryan McDonald, and Joakim Nivre. 2009. *Dependency Parsing*. Morgan and Claypool Publishers.

Kuhlmann, Marco, Carlos Gómez-Rodríguez, and Giorgio Satta. 2011. Dynamic programming algorithms for transition-based dependency parsers. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 673–682, Portland, OR.

Kuhlmann, Marco and Peter Jonsson. 2015. Parsing to noncrossing dependency graphs. *Transactions of the Association for Computational Linguistics*, 3:559–570.

Kuhlmann, Marco and Stephan Oepen. 2016. Towards a catalogue of linguistic graph banks. *Computational Linguistics*, 42(4):819–827.

Lange, Klaus Jörn and Emo Welzl. 1987. String grammars with disconnecting or a basic root of the difficulty in graph grammar parsing. *Discrete Applied Mathematics*, 16(1):17–30.

Lautemann, Clemens. 1988. Decomposition trees: Structured graph representation and efficient algorithms. In *Proceedings of the 13th Colloquium on Trees in Algebra and Programming (CAAP-88)*, pages 28–39, London.

Lautemann, Clemens. 1990. The complexity of graph languages generated by hyperedge replacement. *Acta Informatica*, 27(5):399–421.

Liu, Fei, Jeffrey Flanigan, Sam Thomson, Norman Sadeh, and Noah A. Smith. 2015. Toward abstractive summarization using semantic representations. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1077–1086, Denver, CO.

May, Jonathan. 2016. Semeval-2016 task 8: Meaning representation parsing. In *Proceedings of the 10th International Workshop on Semantic Evaluation (SemEval-2016)*, pages 1063–1073, San Diego, CA.

Nederhof, Mark-Jan and Giorgio Satta. 2004. Tabular parsing. In Carlos Martín-Vide, Victor Mitrana, and Gheorghe Păun, editors, *Formal Languages and Applications*. Springer, Berlin, Heidelberg. pages 529–549.

Oepen, Stephan, Marco Kuhlmann, Yusuke Miyao, Daniel Zeman, Silvie Cinkova, Dan Flickinger, Jan Hajic, and Zdenka Uresova. 2015. Semeval 2015 task 18: Broad-coverage semantic dependency parsing. In *Proceedings of the 9th International Workshop on Semantic Evaluation (SemEval 2015)*, pages 915–926, Denver, CO.

Oepen, Stephan, 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, Dublin.

Peng, Xiaochang, Linfeng Song, and Daniel Gildea. 2015. A synchronous hyperedge replacement grammar based approach for AMR parsing. In *Proceedings of the Nineteenth Conference on Computational Natural Language Learning (CoNLL-15)*, pages 731–739, Beijing.

Peng, Xiaochang, Linfeng Song, Daniel Gildea, and Giorgio Satta. 2018. Sequence-to-sequence models for cache transition systems. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (ACL-18)*, pages 1842–1852, Melbourne.

Pourdamghani, Nima, Yang Gao, Ulf Hermjakob, and Kevin Knight. 2014. Aligning English strings with abstract meaning representation graphs. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 425–429, Doha.

Rao, Sudha, Daniel Marcu, Kevin Knight, and Hal Daumé III. 2017. Biomedical event extraction using abstract meaning representation. In *Workshop on Biomedical Natural Language Processing (BioNLP 2017)*, pages 126–135, Vancouver.

Schluter, Natalie. 2015. The complexity of finding the maximum spanning DAG and other restrictions for DAG parsing of natural language. In *Proceedings of the Fourth Joint Conference on Lexical and Computational Semantics (*SEM 2015)*, pages 259–268, Denver, CO.

Takase, Sho, Jun Suzuki, Naoaki Okazaki, Tsutomu Hirao, and Masaaki Nagata. 2016. Neural headline generation on abstract meaning representation. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 1054–1059, Austin, TX.

Toutanova, Kristina, Aria Haghighi, and Christopher Manning. 2005. Joint learning improves semantic role labeling. *In Proceedings of the 43rd Annual Conference of the Association for Computational Linguistics (ACL-05)*, pages 589–596, Ann Arbor, MI.

Wang, Yanshan, Sijia Liu, Majid Rastegar-Mojarad, Liwei Wang, Feichen Shen, Fei Liu, and Hongfang Liu. 2017. Dependency and AMR embeddings for drug-drug interaction extraction from biomedical literature. In *Proceedings of the 8th ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics*, pages 36–43, Boston, MA.