Parsing Graphs with Hyperedge Replacement Grammars

David Chiang
Information Sciences Institute
University of Southern California

Jacob Andreas
Columbia University
University of Cambridge

Daniel Bauer
Department of Computer Science
Columbia University

Karl Moritz Hermann
Department of Computer Science
University of Oxford

Bevan Jones
University of Edinburgh
Macquarie University

Kevin Knight
Information Sciences Institute
University of Southern California

Abstract

Hyperedge replacement grammar (HRG) is a formalism for generating and transforming graphs that has potential applications in natural language understanding and generation. A recognition algorithm due to Lautemann is known to be polynomial-time for graphs that are connected and of bounded degree. We present a more precise characterization of the algorithm's complexity, an optimization analogous to binarization of context-free grammars, and some important implementation details, resulting in an algorithm that is practical for natural-language applications. The algorithm is part of *Bolinas*, a new software toolkit for HRG processing.

1 Introduction

Hyperedge replacement grammar (HRG) is a context-free rewriting formalism for generating graphs (Drewes et al., 1997), and its synchronous counterpart can be used for transforming graphs to/from other graphs or trees. As such, it has great potential for applications in natural language understanding and generation, and semantics-based machine translation (Jones et al., 2012). Figure 1 shows some examples of graphs for natural-language semantics.

A polynomial-time recognition algorithm for HRGs was described by Lautemann (1990), building on the work of Rozenberg and Welzl (1986) on boundary node label controlled grammars, and others have presented polynomial-time algorithms as well (Mazanek and Minas, 2008; Moot, 2008). Although Lautemann's algorithm is correct and tractable, its presentation is prefaced with the remark: "As we are only interested in distinguishing polynomial time from non-polynomial time, the analysis will be rather crude, and implementation details will be explicated as little as possible." Indeed, the key step of the algorithm, which matches a rule against the input graph, is described at a very high level, so that it is not obvious (for a non-expert in graph algorithms) how to implement it. More importantly, this step as described leads to a time complexity that is polynomial, but potentially of very high degree.

In this paper, we describe in detail a more efficient version of this algorithm and its implementation. We give a more precise complexity analysis in terms of the grammar and the size and maximum degree of the input graph, and we show how to optimize it by a process analogous to binarization of CFGs, following Gildea (2011). The resulting algorithm is practical and is implemented as part of the open-source Bolinas toolkit for hyperedge replacement grammars.

2 Hyperedge replacement grammars

We give a short example of how HRG works, followed by formal definitions.

2.1 Example

Consider a weighted graph language involving just two types of semantic frames (*want* and *believe*), two types of entities (*boy* and *girl*), and two roles (*arg0* and *arg1*). Figure 1 shows a few graphs from this language.

Figure 2 shows how to derive one of these graphs using an HRG. The derivation starts with a single edge labeled with the nonterminal symbol S. The first rewriting step replaces this edge with a subgraph, which we might read as "The

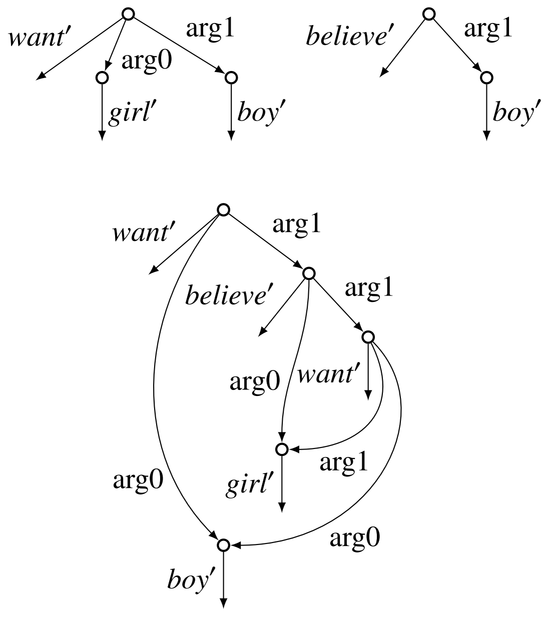


Figure 1: Sample members of a graph language, representing the meanings of (clockwise from upper left): “The girl wants the boy,” “The boy is believed,” and “The boy wants the girl to believe that he wants her.”

boy wants something (X) involving himself.” The second rewriting step replaces the X edge with another subgraph, which we might read as “The boy wants the girl to believe something (Y) involving both of them.” The derivation continues with a third rewriting step, after which there are no more nonterminal-labeled edges.

2.2 Definitions

The graphs we use in this paper have edge labels, but no node labels; while node labels are intuitive for many graphs in NLP, using both node and edge labels complicates the definition of hyperedge grammar and algorithms. All of our graphs are directed (ordered), as the purpose of most graph structures in NLP is to model dependencies between entities.

Definition 1. An *edge-labeled, ordered hypergraph* is a tuple $H = \langle V, E, \ell \rangle$, where

- V is a finite set of nodes
- $E \subseteq V^+$ is a finite set of hyperedges, each of which connects one or more distinct nodes
- $\ell : E \to C$ assigns a label (drawn from the finite set C) to each edge.

For brevity we use the terms *graph* and *hypergraph* interchangeably, and similarly for *edge* and *hyperedge*. In the definition of HRGs, we will use the notion of hypergraph fragments, which are the elementary structures that the grammar assembles into hypergraphs.

Definition 2. A *hypergraph fragment* is a tuple $\langle V, E, \ell, X \rangle$, where $\langle V, E, \ell \rangle$ is a hypergraph and $X \in V^+$ is a list of distinct nodes called the *external nodes*.

The function of graph fragments in HRG is analogous to the right-hand sides of CFG rules and to elementary trees in tree adjoining grammars (Joshi and Schabes, 1997). The external nodes indicate how to integrate a graph into another graph during a derivation, and are analogous to foot nodes. In diagrams, we draw them with a black circle (●).

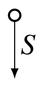
Definition 3. A *hyperedge replacement grammar* (HRG) is a tuple $G = \langle N, T, P, S \rangle$ where

- N and T are finite disjoint sets of nonterminal and terminal symbols
- $S \in N$ is the start symbol
- P is a finite set of productions of the form $A \to R$, where $A \in N$ and R is a graph fragment over $N \cup T$.

We now describe the HRG rewriting mechanism.

Definition 4. Given a HRG G, we define the relation $H \Rightarrow_G H'$ (or, *H' is derived from H in one step*) as follows. Let $e = (v_1 \cdots v_k)$ be an edge in H with label A. Let $(A \to R)$ be a production of G, where R has external nodes $X_R = (u_1 \cdots u_k)$. Then we write $H \Rightarrow_G H'$ if H' is the graph formed by removing e from H, making an isomorphic copy of R, and identifying v_i with (the copy of) u_i for $i = 1, \ldots, k$.

Let $H \Rightarrow_G^* H'$ (or, *H' is derived from H*) be the reflexive, transitive closure of $\Rightarrow_G$. The *graph language* of a grammar G is the (possibly infinite) set of graphs H that have no edges with nonterminal labels such that



$$S \Rightarrow_G^* H.$$

When a HRG rule $(A \to R)$ is applied to an edge e, the mapping of external nodes in R to the

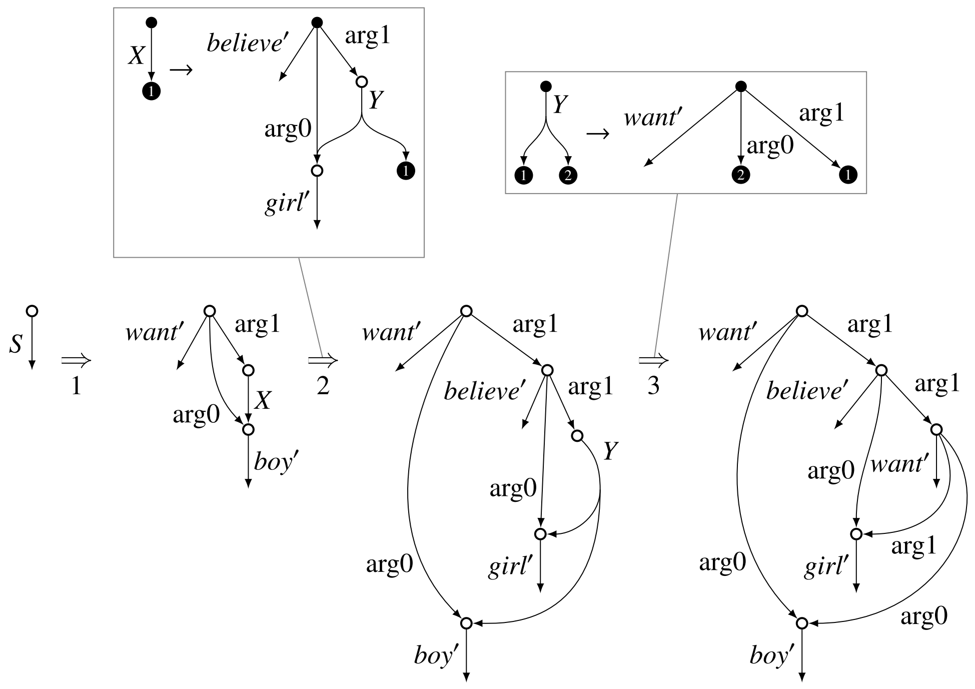


Figure 2: Derivation of a hyperedge replacement grammar for a graph representing the meaning of "The boy wants the girl to believe that he wants her."

nodes of e is implied by the ordering of nodes in e and X_R. When writing grammar rules, we make this ordering explicit by writing the left hand side of a rule as an edge and indexing the external nodes of R on both sides, as shown in Figure 2.

HRG derivations are context-free in the sense that the applicability of each production depends on the nonterminal label of the replaced edge only. This allows us to represent a derivation as a derivation tree, and sets of derivations of a graph as a derivation forest (which can in turn represented as hypergraphs). Thus we can apply many of the methods developed for other context free grammars. For example, it is easy to define weighted and synchronous versions of HRGs.

Definition 5. If K is a semiring, a *K-weighted HRG* is a tuple $G = \langle N, T, P, S, \lambda\rangle$, where $\langle N, T, P, S\rangle$ is a HRG and $\lambda : P \to K$ assigns a weight in K to each production. The weight of a derivation of G is the product of the weights of the productions used in the derivation.

We defer a definition of synchronous HRGs until Section 4, where they are discussed in detail.

3 Parsing

Lautemann's recognition algorithm for HRGs is a generalization of the CKY algorithm for CFGs. Its key step is the matching of a rule against the input graph, analogous to the concatenation of two spans in CKY. The original description leaves open how this matching is done, and because it tries to match the whole rule at once, it has asymptotic complexity exponential in the number of nonterminal edges. In this section, we present a refinement that makes the rule-matching procedure explicit, and because it matches rules little by little, similarly to binarization of CFG rules, it does so more efficiently than the original.

Let H be the input graph. Let n be the number of nodes in H, and d be the maximum degree of any node. Let G be a HRG. For simplicity, we assume that the right-hand sides of rules are connected. This restriction entails that each graph generated by G is connected; therefore, we assume that H is connected as well. Finally, let m be an arbitrary node of H called the *marker node*, whose usage will become clear below.[1]

3.1 Representing subgraphs

Just as CKY deals with substrings $(i, j]$ of the input, the HRG parsing algorithm deals with edge-induced subgraphs I of the input. An edge-induced subgraph of $H = \langle V, E, \ell\rangle$ is, for some

[1]To handle the more general case where H is not connected, we would need a marker for each component.

subset $E' \subseteq E$, the smallest subgraph containing all edges in E'. From now on, we will assume that all subgraphs are edge-induced subgraphs.

In CKY, the two endpoints i and j completely specify the recognized part of the input, $w_{i+1} \cdots w_j$. Likewise, we do not need to store all of I explicitly.

Definition 6. Let I be a subgraph of H. A *boundary node* of I is a node in I which is either a node with an edge in $H \setminus I$ or an external node. A *boundary edge* of I is an edge in I which has a boundary node as an endpoint. The *boundary representation* of I is the tuple $\langle bn(I), be(I, v), m \in I \rangle$, where

- $bn(I)$ is the set of boundary nodes of I
- $be(I, v)$ be the set of boundary edges of v in I
- $(m \in I)$ is a flag indicating whether the marker node is in I.

The boundary representation of I suffices to specify I compactly.

Proposition 1. *If I and I' are two subgraphs of H with the same boundary representation, then $I = I'$.*

Proof. Case 1: $bn(I)$ is empty. If $m \in I$ and $m \in I'$, then all edges of H must belong to both I and I', that is, $I = I' = H$. Otherwise, if $m \notin I$ and $m \notin I'$, then no edges can belong to either I or I', that is, $I = I' = \emptyset$.

Case 2: $bn(I)$ is nonempty. Suppose $I \neq I'$; without loss of generality, suppose that there is an edge e that is in $I \setminus I'$. Let π be the shortest path (ignoring edge direction) that begins with e and ends with a boundary node. All the edges along π must be in $I \setminus I'$, or else there would be a boundary node in the middle of π, and π would not be the shortest path from e to a boundary node. Then, in particular, the last edge of π must be in $I \setminus I'$. Since it has a boundary node as an endpoint, it must be a boundary edge of I, but cannot be a boundary edge of I', which is a contradiction. □

If two subgraphs are disjoint, we can use their boundary representations to compute the boundary representation of their union.

Proposition 2. *Let I and J be two subgraphs whose edges are disjoint. A node v is a boundary node of $I \cup J$ iff one of the following holds:*

(i) v is a boundary node of one subgraph but not the other

(ii) v is a boundary node of both subgraphs, and has an edge which is not a boundary edge of either.

An edge is a boundary edge of $I \cup J$ iff it has a boundary node of $I \cup J$ as an endpoint and is a boundary edge of I or J.

Proof. ($\Rightarrow$) v has an edge in either I or J and an edge e outside both I and J. Therefore it must be a boundary node of either I or J. Moreover, e is not a boundary edge of either, satisfying condition (ii).

($\Leftarrow$) Case (i): without loss of generality, assume v is a boundary node of I. It has an edge e in I, and therefore in $I \cup J$, and an edge e' outside I, which must also be outside J. For $e \notin J$ (because I and J are disjoint), and if $e' \in J$, then v would be a boundary node of J. Therefore, $e' \notin I \cup J$, so v is a boundary node of $I \cup J$. Case (ii): v has an edge in I and therefore $I \cup J$, and an edge not in either I or J. □

This result leads to Algorithm 1, which runs in time linear in the number of boundary nodes.

Algorithm 1 Compute the union of two disjoint subgraphs I and J.

```
for all $v \in bn(I)$ do
    $E \leftarrow be(I, v) \cup be(J, v)$
    if $v \notin bn(J)$ or $v$ has an edge not in $E$ then
        add $v$ to $bn(I \cup J)$
        $be(I \cup J, v) \leftarrow E$
for all $v \in bn(J)$ do
    if $v \notin bn(I)$ then
        add $v$ to $bn(I \cup J)$
        $be(I \cup J, v) \leftarrow be(I, v) \cup be(J, v)$
$(m \in I \cup J) \leftarrow (m \in I) \vee (m \in J)$
```

In practice, for small subgraphs, it may be more efficient simply to use an explicit set of edges instead of the boundary representation. For the GeoQuery corpus (Tang and Mooney, 2001), whose graphs are only 7.4 nodes on average, we generally find this to be the case.

3.2 Treewidth

Lautemann's algorithm tries to match a rule against the input graph all at once. But we can optimize the algorithm by matching a rule incrementally. This is analogous to the rank-minimization problem for linear context-free rewriting systems. Gildea has shown that this problem is related to the notion of *treewidth* (Gildea, 2011), which we review briefly here.

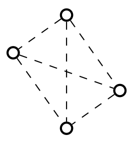
Definition 7. A *tree decomposition* of a graph $H = \langle V, E \rangle$ is a tree T, each of whose nodes η is associated with sets $V_\eta \subseteq V$ and $E_\eta \subseteq E$, with the following properties:

1. Vertex cover: For each $v \in V$, there is a node $\eta \in T$ such that $v \in V_\eta$.
2. Edge cover: For each $e = (v_1 \cdots v_k) \in E$, there is exactly one node $\eta \in T$ such that $e \in E_\eta$. We say that η *introduces* e. Moreover, $v_1, \ldots, v_k \in V_\eta$.
3. Running intersection: For each $v \in V$, the set $\{\eta \in T \mid v \in V_\eta\}$ is connected.

The *width* of T is $\max |V_\eta| - 1$. The *treewidth* of H is the minimal width of any tree decomposition of H.

A tree decomposition of a graph fragment $\langle V, E, X \rangle$ is a tree decomposition of $\langle V, E \rangle$ that has the additional property that all the external nodes belong to V_η for some η. (Without loss of generality, we assume that η is the root.)

For example, Figure 3b shows a graph, and Figure 3c shows a tree decomposition. This decomposition has width three, because its largest node has 4 elements. In general, a tree has width one, and it can be shown that a graph has treewidth at most two iff it does not have the following graph as a minor (Bodlaender, 1997):

$K_4 =$ 

Finding a tree decomposition with minimal width is in general NP-hard (Arnborg et al., 1987). However, we find that for the graphs we are interested in in NLP applications, even a naïve algorithm gives tree decompositions of low width in practice: simply perform a depth-first traversal of the edges of the graph, forming a tree T. Then, augment the V_η as necessary to satisfy the running intersection property.

As a test, we extracted rules from the GeoQuery corpus (Tang and Mooney, 2001) using the SynSem algorithm (Jones et al., 2012), and computed tree decompositions exactly using a branch-and-bound method (Gogate and Dechter, 2004) and this approximate method. Table 1 shows that, in practice, treewidths are not very high even when computed only approximately.

method	mean	max
exact	1.491	2
approximate	1.494	3

Table 1: Mean and maximum treewidths of rules extracted from the GeoQuery corpus, using exact and approximate methods.

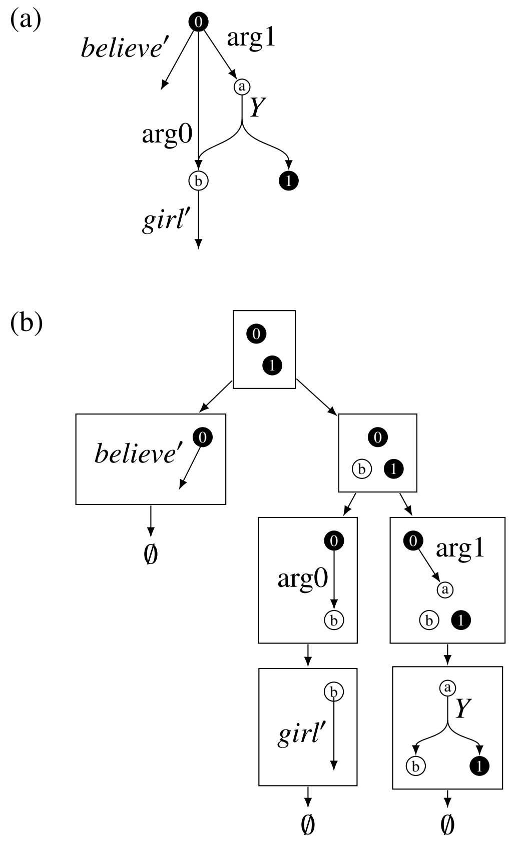


Figure 3: (a) A rule right-hand side, and (b) a nice tree decomposition.

Any tree decomposition can be converted into one which is *nice* in the following sense (simplified from Cygan et al. (2011)). Each tree node η must be one of:

- A leaf node, such that $V_\eta = \emptyset$.
- A unary node, which introduces exactly one edge e.
- A binary node, which introduces no edges.

The example decomposition in Figure 3c is nice. This canonical form simplifies the operation of the parser described in the following section.

Let G be a HRG. For each production $(A \rightarrow R) \in G$, find a nice tree decomposition of R and call it T_R. The treewidth of G is the maximum treewidth of any right-hand side in G.

The basic idea of the recognition algorithm is to recognize the right-hand side of each rule incrementally by working bottom-up on its tree decomposition. The properties of tree decomposition allow us to limit the number of boundary nodes of the partially-recognized rule.

More formally, let $R_{\trianglerighteq\eta}$ be the subgraph of R induced by the union of $E_{\eta'}$ for all η' equal to or dominated by η. Then we can show the following.

Proposition 3. *Let R be a graph fragment, and assume a tree decomposition of R. All the boundary nodes of $R_{\trianglerighteq\eta}$ belong to $V_\eta \cap V_{parent(\eta)}$.*

Proof. Let v be a boundary node of $R_{\trianglerighteq\eta}$. Node v must have an edge in $R_{\trianglerighteq\eta}$ and therefore in $R_{\eta'}$ for some η' dominated by or equal to η.

Case 1: v is an external node. Since the root node contains all the external nodes, by the running intersection property, both V_η and $V_{parent(\eta)}$ must contain v as well.

Case 2: v has an edge not in $R_{\trianglerighteq\eta}$. Therefore there must be a tree node not dominated by or equal to η that contains this edge, and therefore v. So by the running intersection property, η and its parent must contain v as well. □

This result, in turn, will allow us to bound the complexity of the parsing algorithm in terms of the treewidth of G.

3.3 Inference rules

We present the parsing algorithm as a deductive system (Shieber et al., 1995). The items have one of two forms. A *passive* item has the form $[A, I, X]$, where $X \in V^+$ is an explicit ordering of the boundary nodes of I. This means that we have recognized that $A \Rightarrow^*_G I$. Thus, the goal item is $[S, H, \epsilon]$. An *active* item has the form $[A \rightarrow R, \eta, I, \phi]$, where

- $(A \rightarrow R)$ is a production of G
- η is a node of T_R
- I is a subgraph of H
- ϕ is a bijection between the boundary nodes of $R_{\trianglerighteq\eta}$ and those of I.

The parser must ensure that ϕ is a bijection when it creates a new item. Below, we use the notation $\{e \mapsto e'\}$ or $\{e \mapsto X\}$ for the mapping that sends each node of e to the corresponding node of e' or X.

Passive items are generated by the following rule:

- Root
$$\frac{[B \rightarrow Q, \theta, J, \psi]}{[B, J, X]}$$
where θ is the root of T_Q, and $X_j = \psi(X_{Q,j})$.

If we assume that the T_R are nice, then the inference rules that generate active items follow the different types of nodes in a nice tree decomposition:

- Leaf
$$\frac{}{[A \rightarrow R, \eta, \emptyset, \emptyset]}$$
where η is a leaf node of T_R.
- (Unary) Nonterminal
$$\frac{[A \rightarrow R, \eta_1, I, \phi] \quad [B, J, X]}{[A \rightarrow R, \eta, I \cup J, \phi \cup \{e \mapsto X\}]}$$
where η_1 is the only child of η, and e is introduced by η and is labeled with nonterminal B.
- (Unary) Terminal
$$\frac{[A \rightarrow R, \eta_1, I, \phi]}{[A \rightarrow R, \eta, I \cup \{e'\}, \phi \cup \{e \mapsto e'\}]}$$
where η_1 is the only child of η, e is introduced by η, and e and e' are both labeled with terminal a.
- Binary
$$\frac{[A \rightarrow R, \eta_1, I, \phi_1] \quad [A \rightarrow R, \eta_2, J, \phi_2]}{[A \rightarrow R, \eta, I \cup J, \phi_1 \cup \phi_2]}$$
where η_1 and η_2 are the two children of η.

In the Nonterminal, Terminal, and Binary rules, we form unions of subgraphs and unions of mappings. When forming the union of two subgraphs, we require that the subgraphs be disjoint (however, see Section 3.4 below for a relaxation of this condition). When forming the union of two mappings, we require that the result be a bijection. If either of these conditions is not met, the inference rule cannot apply.

For efficiency, it is important to index the items for fast access. For the Nonterminal inference rule, passive items $[B, J, X]$ should be indexed by key $\langle B, |bn(J)|\rangle$, so that when the next item on the agenda is an active item $[A \rightarrow R, \eta_1, I, \phi]$, we know that all possible matching passive items are

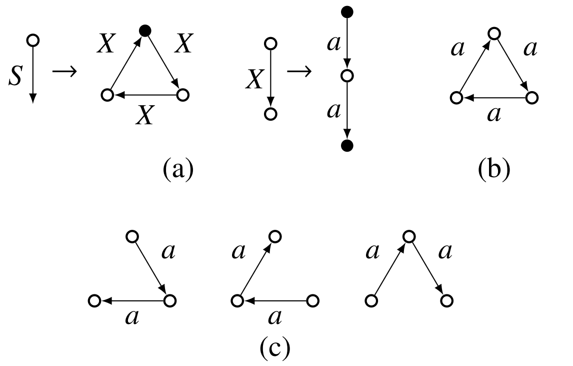


Figure 4: Illustration of unsoundness in the recognition algorithm without the disjointness check. Using grammar (a), the recognition algorithm would incorrectly accept the graph (b) by assembling together the three overlapping fragments (c).

under key $\langle \ell(e), |e| \rangle$. Similarly, active items should be indexed by key $\langle \ell(e), |e| \rangle$ so that they can be found when the next item on the agenda is a passive item. For the Binary inference rule, active items should be indexed by their tree node (η_1 or η_2).

This procedure can easily be extended to produce a packed forest of all possible derivations of the input graph, representable as a hypergraph just as for other context-free rewriting formalisms. The Viterbi algorithm can then be applied to this representation to find the highest-probability derivation, or the Inside/Outside algorithm to set weights by Expectation-Maximization.

3.4 The disjointness check

A successful proof using the inference rules above builds an HRG derivation (comprising all the rewrites used by the Nonterminal rule) which derives a graph H', as well as a graph isomorphism $\phi : H' \to H$ (the union of the mappings from all the items).

During inference, whenever we form the union of two subgraphs, we require that the subgraphs be disjoint. This is a rather expensive operation: it can be done using only their boundary representations, but the best algorithm we are aware of is still quadratic in the number of boundary nodes.

Is it possible to drop the disjointness check? If we did so, it would become possible for the algorithm to recognize the same part of H twice. For example, Figure 4 shows an example of a grammar and an input that would be incorrectly recognized.

However, we can replace the disjointness check with a weaker and faster check such that any derivation that merges two non-disjoint subgraphs will ultimately fail, and therefore the derived graph H' is isomorphic to the input graph H' as desired. This weaker check is to require, when merging two subgraphs I and J, that:

1. I and J have no boundary edges in common, and

2. If m belongs to both I and J, it must be a boundary node of both.

Condition (1) is enough to guarantee that ϕ is *locally one-to-one* in the sense that for all $v \in H'$, ϕ restricted to v and its neighbors is one-to-one. This is easy to show by induction: if $\phi_I : I' \to H$ and $\phi_J : J' \to H$ are locally one-to-one, then $\phi_I \cup \phi_J$ must also be, provided condition (1) is met. Intuitively, the consequence of this is that we can detect any place where ϕ changes (say) from being one-to-one to two-to-one. So if ϕ is two-to-one, then it must be two-to-one everywhere (as in the example of Figure 4).

But condition (2) guarantees that ϕ maps only one node to the marker m. We can show this again by induction: if ϕ_I and ϕ_J each map only one node to m, then $\phi_I \cup \phi_J$ must map only one node to m, by a combination of condition (2) and the fact that the inference rules guarantee that ϕ_I, ϕ_J, and $\phi_I \cup \phi_J$ are one-to-one on boundary nodes.

Then we can show that, since m is recognized exactly once, the whole graph is also recognized exactly once.

Proposition 4. *If H and H' are connected graphs, $\phi : H' \to H$ is locally one-to-one, and ϕ^{-1} is defined for some node of H, then ϕ is a bijection.*

Proof. Suppose that ϕ is not a bijection. Then there must be two nodes $v'_1, v'_2 \in H'$ such that $\phi(v'_1) = \phi(v'_2) = v \in H$. We also know that there is a node, namely, m, such that $m' = \phi^{-1}(m)$ is defined.[2] Choose a path π (ignoring edge direction) from v to m. Because ϕ is a local isomorphism, we can construct a path from v'_1 to m' that maps to π. Similarly, we can construct a path from v'_2 to m' that maps to π. Let u' be the first node that these two paths have in common. But u' must have two edges that map to the same edge, which is a contradiction. □

[2]If H were not connected, we would choose the marker in the same connected component as v.

3.5 Complexity

The key to the efficiency of the algorithm is that the treewidth of G leads to a bound on the number of boundary nodes we must keep track of at any time.

Let k be the treewidth of G. The time complexity of the algorithm is the number of ways of instantiating the inference rules. Each inference rule mentions only boundary nodes of $R_{\trianglerighteq\eta}$ or $R_{\trianglerighteq\eta_i}$, all of which belong to V_η (by Proposition 3), so there are at most $|V_\eta| \leq k + 1$ of them. In the Nonterminal and Binary inference rules, each boundary edge could belong to I or J or neither. Therefore, the number of possible instantiations of any inference rule is in $O((3^d n)^{k+1})$.

The space complexity of the algorithm is the number of possible items. For each active item $[A \to R, \eta, I, \phi]$, every boundary node of $R_{\trianglerighteq\eta}$ must belong to $V_\eta \cap V_{parent(\eta)}$ (by Proposition 3). Therefore the number of boundary nodes is at most $k+1$ (but typically less), and the number of possible items is in $O((2^d n)^{k+1})$.

4 Synchronous Parsing

As mentioned in Section 2.2, because HRGs have context-free derivation trees, it is easy to define *synchronous HRGs*, which define mappings between languages of graphs.

Definition 8. A *synchronous hyperedge replacement grammar* (SHRG) is a tuple $G = \langle N, T, T', P, S\rangle$, where

- N is a finite set of nonterminal symbols
- T and T' are finite sets of terminal symbols
- $S \in N$ is the start symbol
- P is a finite set of productions of the form $(A \to \langle R, R', \sim\rangle)$, where R is a graph fragment over $N \cup T$ and R' is a graph fragment over $N \cup T'$. The relation $\sim$ is a bijection linking nonterminal mentions in R and R', such that if $e \sim e'$, then they have the same label. We call R the *source* side and R' the *target* side.

Some NLP applications (for example, word alignment) require *synchronous parsing*: given a pair of graphs, finding the derivation or forest of derivations that simultaneously generate both the source and target. The algorithm to do this is a straightforward generalization of the HRG parsing algorithm. For each rule $(A \to \langle R, R', \sim\rangle)$, we construct a nice tree decomposition of $R \cup R'$ such that:

- All the external nodes of both R and R' belong to V_η for some η. (Without loss of generality, assume that η is the root.)
- If $e \sim e'$, then e and e' are introduced by the same tree node.

In the synchronous parsing algorithm, passive items have the form $[A, I, X, I', X']$ and active items have the form $[A \to R : R', \eta, I, \phi, I', \phi']$. For brevity we omit a re-presentation of all the inference rules, as they are very similar to their non-synchronous counterparts. The main difference is that in the Nonterminal rule, two linked edges are rewritten simultaneously:

$$\frac{[A \to R : R', \eta_1, I, \phi, I', \phi'] \quad [B, J, X, J', X']}{\begin{array}{r}[A \to R : R', \eta, I \cup J, \phi \cup \{e_j \mapsto X_j\}, \\ I' \cup J', \phi' \cup \{e'_j \mapsto X'_j\}]\end{array}}$$

where η_1 is the only child of η, e and e' are both introduced by η and $e \sim e'$, and both are labeled with nonterminal B.

The complexity of the parsing algorithm is again in $O((3^d n)^{k+1})$, where k is now the maximum treewidth of the dependency graph as defined in this section. In general, this treewidth will be greater than the treewidth of either the source or target side on its own, so that synchronous parsing is generally slower than standard parsing.

5 Conclusion

Although Lautemann's polynomial-time extension of CKY to HRGs has been known for some time, the desire to use graph grammars for large-scale NLP applications introduces some practical considerations not accounted for in Lautemann's original presentation. We have provided a detailed description of our refinement of his algorithm and its implementation. It runs in $O((3^d n)^{k+1})$ time and requires $O((2^d n)^{k+1})$ space, where n is the number of nodes in the input graph, d is its maximum degree, and k is the maximum treewidth of the rule right-hand sides in the grammar. We have also described how to extend this algorithm to synchronous parsing. The parsing algorithms described in this paper are implemented in the Bolinas toolkit.[3]

[3]The Bolinas toolkit can be downloaded from ⟨`http://www.isi.edu/licensed-sw/bolinas/`⟩.

Acknowledgements

We would like to thank the anonymous reviewers for their helpful comments. This research was supported in part by ARO grant W911NF-10-1-0533.

References

Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski. 1987. Complexity of finding embeddings in a k-tree. *SIAM Journal on Algebraic and Discrete Methods*, 8(2).

Hans L. Bodlaender. 1997. Treewidth: Algorithmic techniques and results. In *Proc. 22nd International Symposium on Mathematical Foundations of Computer Science (MFCS '97)*, pages 29–36, Berlin. Springer-Verlag.

Marek Cygan, Jesper Nederlof, Marcin Pilipczuk, Michał Pilipczuk, Johan M. M. van Rooij, and Jakub Onufry Wojtaszczyk. 2011. Solving connectivity problems parameterized by treewidth in single exponential time. *Computing Research Repository*, abs/1103.0534.

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

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

Vibhav Gogate and Rina Dechter. 2004. A complete anytime algorithm for treewidth. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*.

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

Aravind K. Joshi and Yves Schabes. 1997. Tree-adjoining grammars. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages and Automata*, volume 3, pages 69–124. Springer.

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

Steffen Mazanek and Mark Minas. 2008. Parsing of hyperedge replacement grammars with graph parser combinators. In *Proc. 7th International Workshop on Graph Transformation and Visual Modeling Techniques*.

Richard Moot. 2008. Lambek grammars, tree adjoining grammars and hyperedge replacement grammars. In *Proc. TAG+9*, pages 65–72.

Grzegorz Rozenberg and Emo Welzl. 1986. Boundary NLC graph grammars—basic definitions, normal forms, and complexity. *Information and Control*, 69:136–167.

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

Lappoon Tang and Raymond Mooney. 2001. Using multiple clause constructors in inductive logic programming for semantic parsing. In *Proc. European Conference on Machine Learning*.