Efficient techniques for parsing with tree automata

Jonas Groschwitz and **Alexander Koller**
Department of Linguistics
University of Potsdam
Germany
groschwi|akoller@uni-potsdam.de

Mark Johnson
Department of Computing
Macquarie University
Australia
mark.johnson@mq.edu.au

Abstract

Parsing for a wide variety of grammar formalisms can be performed by intersecting finite tree automata. However, naive implementations of parsing by intersection are very inefficient. We present techniques that speed up tree-automata-based parsing, to the point that it becomes practically feasible on realistic data when applied to context-free, TAG, and graph parsing. For graph parsing, we obtain the best runtimes in the literature.

1 Introduction

Grammar formalisms that go beyond context-free grammars have recently enjoyed renewed attention throughout computational linguistics. Classical grammar formalisms such as TAG (Joshi and Schabes, 1997) and CCG (Steedman, 2001) have been equipped with expressive statistical models, and high-performance parsers have become available (Clark and Curran, 2007; Lewis and Steedman, 2014; Kallmeyer and Maier, 2013). Synchronous grammar formalisms such as synchronous context-free grammars (Chiang, 2007) and tree-to-string transducers (Galley et al., 2004; Graehl et al., 2008; Seemann et al., 2015) are being used as models that incorporate syntactic information in statistical machine translation. Synchronous string-to-tree (Wong and Mooney, 2006) and string-to-graph grammars (Chiang et al., 2013) have been applied to semantic parsing; and so forth.

Each of these grammar formalisms requires its users to develop new algorithms for parsing and training. This comes with challenges that are both practical and theoretical. From a theoretical perspective, many of these algorithms are basically the same, in that they rest upon a CKY-style parsing algorithm which recursively explores substructures of the input object and assigns them nonterminal symbols, but their exact relationship is rarely made explicit. On the practical side, this parsing algorithm and its extensions (e.g. to EM training) have to be implemented and optimized from scratch for each new grammar formalism. Thus, development time is spent on reinventing wheels that are slightly different from previous ones, and the resulting implementations still tend to underperform.

Koller and Kuhlmann (2011) introduced *Interpreted Regular Tree Grammars (IRTGs)* in order to address this situation. An IRTG represents a language by describing a regular language of derivation trees, each of which is mapped to a term over some algebra and evaluated there. Grammars from a wide range of monolingual and synchronous formalisms can be mapped into IRTGs by using different algebras: Context-free and tree-adjoining grammars use string algebras of different kinds, graph grammars can be captured by using graph algebras, and so on. In addition, IRTGs come with a universal parsing algorithm based on closure results for tree automata. Implementing and optimizing this parsing algorithm once, one could apply it to all grammar formalisms that can be mapped to IRTG. However, while Koller and Kuhlmann show that asymptotically optimal parsing is possible in theory, it is non-trivial to implement their algorithm optimally.

In this paper, we introduce practical algorithms for the two key operations underlying IRTG parsing: computing the intersection of two tree automata and applying an inverse tree homomorphism to a tree automaton. After defining IRTGs (Section 2), we will first illustrate that a naive bottom-up implementation of the intersection algorithm yields asymptotic parsing complexities that are too high (Section 3). We will then

show how the parsing complexity can be improved by combining algebra-specific index data structures with a generic parsing algorithm (Section 4), and by replacing bottom-up with top-down queries (Section 5). In contrast to the naive algorithm, both of these methods achieve the expected asymptotic complexities, e.g. $O(n^3)$ for context-free parsing, $O(n^6)$ for TAG parsing, etc. Furthermore, an evaluation with realistic grammars shows that our algorithms improve practical parsing times with IRTG grammars encoding context-free grammars, tree-adjoining grammars, and graph grammars by orders of magnitude (Section 6). Thus our algorithms make IRTG parsing practically feasible for the first time; for graph parsing, we obtain the fastest reported runtimes.

2 Interpreted Regular Tree Grammars

We will first define IRTGs and explain how the universal parsing algorithm for IRTGs works.

2.1 Formal foundations

First, we introduce some fundamental theoretical concepts and notation.

A *signature* Σ is a finite set of symbols $r, f, \ldots$, each of which has an *arity* $\mathsf{ar}(r) \geq 0$. A *tree* t over the signature Σ is a term of the form $r(t_1, \ldots, t_n)$, where the t_i are trees and $r \in \Sigma$ has arity n. We identify the nodes of t by their Gorn addresses, i.e. paths $\pi \in \mathbb{N}^*$ from the root to the node, and write $t(\pi)$ for the label of π. We write T_Σ for the set of all trees over Σ, and $T_\Sigma(\mathcal{X}_k)$ for the trees in which each node either has a label from Σ, or is a leaf labeled with one of the variables $\{x_1, \ldots, x_k\}$.

A *(linear, nondeleting) tree homomorphism* h from a signature Σ to a signature Δ is a mapping $h : T_\Sigma \to T_\Delta$. It is defined by specifying, for each symbol $r \in \Sigma$ of arity k, a term $h(r) \in T_\Delta(\mathcal{X}_k)$ in which each variable occurs exactly once. This symbol-wise mapping is lifted to entire trees by letting $h(r(t_1, \ldots, t_k)) = h(r)[h(t_1), \ldots, h(t_k)]$, i.e. by replacing the variable x_i in $h(r)$ by the recursively computed value $h(t_i)$.

Let Δ be a signature. A Δ-*algebra* $\mathcal{A}$ consists of a nonempty set A, called the *domain*, and for each symbol $f \in \Delta$ with arity k, a function $f^{\mathcal{A}} : A^k \to A$, the *operation* associated with f. We can *evaluate* any term $t \in T_\Delta$ to a value $t^{\mathcal{A}} \in A$, by evaluating the operation symbols bottom-up. In this paper, we will be particularly interested in the *string algebra* E^* over the finite alphabet E. Its domain is the set of all strings over E. For each symbol $a \in E$, it has a nullary operation symbol a with $\mathsf{a}^{E^*} = a$. It also has a single binary operation symbol $*$, such that $*^{E^*}(w_1, w_2)$ is the concatenation of the strings w_1 and w_2. Thus the term $*(\mathsf{John}, *(\mathsf{walks}, *(\mathsf{on}, \mathsf{Mars})))$ in Fig. 2b evaluates to the string "John walks on Mars".

automaton rule	homomorphism
$\mathsf{S} \to \mathsf{r}_1(\mathsf{NP}, \mathsf{VP})$	$*(x_1, x_2)$
$\mathsf{NP} \to \mathsf{r}_2$	John
$\mathsf{VP} \to \mathsf{r}_3$	walks
$\mathsf{VP} \to \mathsf{r}_4(\mathsf{VP}, \mathsf{NP})$	$*(x_1, *(\mathsf{on}, x_2))$
$\mathsf{NP} \to \mathsf{r}_5$	Mars

Figure 1: An example IRTG.

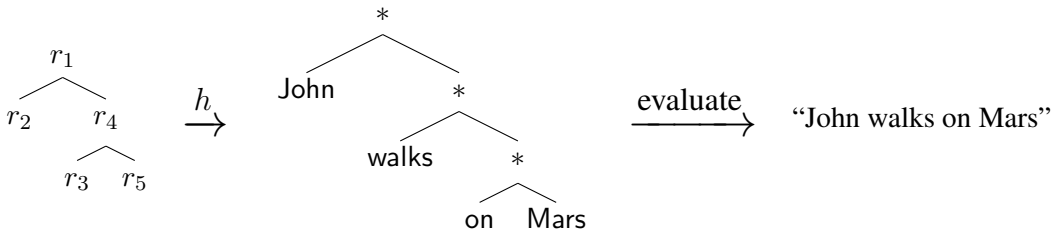
A *finite tree automaton* M over the signature Σ is a structure $M = (\Sigma, Q, R, X_F)$, where Q is a finite set of *states* and $X_F \in Q$ is a *final state*. R is a finite set of *transition rules* of the form $X \to r(X_1, \ldots, X_k)$, where the *terminal symbol* $r \in \Sigma$ is of arity k and $X, X_1, \ldots, X_k \in Q$. A tree automaton can *run* non-deterministically on a tree $t \in T_\Sigma$ by assigning states to the nodes of t bottom-up. If we have $t = r(t_1, \ldots, t_n)$ and M can assign the state X_i to each t_i, written $X_i \to^* t_i$, then we also have $X \to^* t$. We say that M *accepts* t if $X_F \to^* t$, and define the *language* $L(M) \subseteq T_\Sigma$ of M as the (possibly infinite) set of all trees that M accepts. An example of a tree automaton (with states S, NP, etc.) is shown in the "automaton rule" column of Fig. 1. It accepts, among others, the tree τ_1 in Fig. 2a.

Tree automata can be defined top-down or bottom-up, and are equivalent to *regular tree grammars*. The languages that can be accepted by finite tree automata are called the *regular tree languages*. See e.g. Comon et al. (2008) for details.

2.2 Interpreted regular tree grammars

We can combine tree automata, homomorphisms, and algebras into grammars that can describe languages of arbitrary objects, as well as relations between such objects – in a way that inherits many technical properties from context-free grammars, while extending the expressive capacity.

An *interpreted regular tree grammar* (IRTG, Koller and Kuhlmann (2011)) $\mathcal{G} = (M, (h_1, \mathcal{A}_1), \ldots, (h_n, \mathcal{A}_n))$ consists of a tree automaton M over some signature Σ, together with an arbitrary number n of *inter-*



(a) Tree τ_1. (b) Term $h(\tau_1)$. (c) $h(\tau_1)$ evaluated in E^*.

Figure 2: The tree τ_1, evaluated by the homomorphism h and the algebra E^*

pretations $(h_i, \mathcal{A}_i)$, where each $\mathcal{A}_i$ is an algebra over some signature Δ_i and each h_i is a tree homomorphism from Σ to Δ_i. The automaton M describes a language $L(M)$ of *derivation trees* which represent abstract syntactic structures. Each derivation tree τ is then interpreted n ways: we map it to a term $h_i(\tau) \in T_{\Delta_i}$, and then we evaluate $h_i(\tau)$ to a value $a_i = h_i(\tau)^{\mathcal{A}_i} \in A_i$ of the algebra $\mathcal{A}_i$. Thus, the IRTG $\mathcal{G}$ defines a language $L(\mathcal{G}) = \{(h_1(\tau)^{\mathcal{A}_1}, \ldots, h_n(\tau)^{\mathcal{A}_n}) \mid \tau \in L(M)\}$, which is an n-place relation between the domains of the algebras.

Consider the IRTG $\mathcal{G}$ shown in Fig. 1. The "automaton rule" column indicates the five rules of M; the final state is S. We already saw the derivation tree $\tau_1 \in L(M)$. $\mathcal{G}$ has a single interpretation, into a string algebra E^*, and with a homomorphism that is specified by the "homomorphism" column; for instance, $h(\mathsf{r}_1) = *(x_1, x_2)$ and $h(\mathsf{r}_2) = \mathsf{John}$. Applying this homomorphism to τ_1, we obtain the term $h(\tau_1)$ in Fig. 2b. As we saw earlier, this term evaluates in the string algebra to the string "John walks on Mars" (Fig. 2c). Thus this string is an element of $L(\mathcal{G})$.

We assume that no two rules of M use the same terminal symbol; this is generally not required in tree automata, but every IRTG can be brought into this convenient form. Furthermore, we focus (but only for simplicity of presentation) on IRTGs that use a single string-algebra interpretation, as in Fig. 1. Such grammars capture context-free grammars. However, IRTGs can capture a wide variety of grammar formalisms by using different algebras. For instance, an interpretation that uses a TAG string algebra (or TAG derived-tree algebra) models a tree-adjoining grammar (Koller and Kuhlmann, 2012), and an interpretation into an s-graph algebra models a hyperedge replacement graph grammar (HRG, Groschwitz et al. (2015)). By using multiple algebras, IRTGs can also represent synchronous grammars and (bottom-up) tree-to-tree and tree-to-string transducers. In general, any grammar formalism whose grammars describe derivations in terms of a finite set of states can typically be converted into IRTG.

2.3 Parsing IRTGs

Koller and Kuhlmann (2011) present a uniform parsing algorithm for IRTGs based on tree automata. The (monolingual) *parsing problem* of IRTG consists in determining, for an IRTG $\mathcal{G}$ and an input object $a \in A$, a representation of the set $\mathsf{parses}(a) = \{\tau \in L(M) \mid h(\tau)^{\mathcal{A}} = a\}$, i.e. of the derivation trees that are grammatically correct and are mapped to a by the interpretation. In the example, we have $\mathsf{parses}(\text{"John walks on Mars"}) = \{\tau_1\}$, where τ_1 is as above. In general, $\mathsf{parses}(a)$ may be infinite, and thus we aim to represent it using a tree automaton Ch_a with $L(\mathrm{Ch}_a) = \mathsf{parses}(a)$, the *parse chart* of a.

We can compute Ch_a as follows. First, observe that $\mathsf{parses}(a) = L(M) \cap h^{-1}(\mathsf{terms}(a))$, where $h^{-1}(L) = \{\tau \in T_\Sigma \mid h(\tau) \in L\}$ (the *inverse homomorphic image*, or *invhom*, of L) and $\mathsf{terms}(a) = \{t \in T_\Delta \mid t^{\mathcal{A}} = a\}$, i.e. the set of all terms that evaluate to a. Now assume that the algebra $\mathcal{A}$ is *regularly decomposable*, which means that every $a \in A$ has a *decomposition automaton* D_a, i.e. there is a tree automaton D_a such that $L(D_a) = \mathsf{terms}(a)$. Because regular tree languages are closed under invhom and intersection, we can then compute a tree automaton Ch_a by intersecting M with the invhom of D_a.

To illustrate the IRTG parsing algorithm, let us compute a chart for the sentence $s =$ "John walks on Mars" with the example grammar $\mathcal{G}$ of Fig. 1. The states of the decomposition automaton D_s are spans $[i, k]$ of s; the final state is $X_F = [1, 5]$. The automaton contains fourteen rules, including the ones shown in Fig. 3a.

$[1,5] \to *([1,2],[2,5])$
$[2,5] \to *([2,3],[3,5])$
$[3,5] \to *([3,4],[4,5])$
$[3,4] \to \mathsf{on}$
$[4,5] \to \mathsf{Mars}$

(a) Some rules of D_s.

$[1,5] \to \mathsf{r}_1([1,2],[2,5])$
$[2,5] \to \mathsf{r}_4([2,3],[4,5])$
$[2,4] \to \mathsf{r}_1([2,3],[3,4])$
$[1,2] \to \mathsf{r}_2$
$[2,3] \to \mathsf{r}_3$

(b) Some rules of $I = h^{-1}(D_s)$.

$\mathsf{S}[1,5] \to \mathsf{r}_1(\mathsf{NP}[1,2],\mathsf{VP}[2,5])$
$\mathsf{VP}[2,5] \to \mathsf{r}_4(\mathsf{VP}[2,3],\mathsf{NP}[4,5])$
$\mathsf{NP}[1,2] \to \mathsf{r}_2$
$\mathsf{VP}[2,3] \to \mathsf{r}_3$
$\mathsf{NP}[4,5] \to \mathsf{r}_5$

(c) The parse chart Ch_s.

Figure 3: Example rules for the sentence $s =$ "John walks on Mars"

Algorithm 1 Naive bottom-up intersection

```
1:  initialize agenda with state pairs for constants
2:  initialize P as empty
3:  while agenda is not empty do
4:    T'X' ← pop(agenda)
5:    add T'X' to P
6:    for T''X'' ∈ P do
7:      for {T1X1, T2X2} = {T'X', T''X''} do
8:        for T → r(T1, T2) in M_L do
9:          for X → r(X1, X2) in M_R do
10:           store TX → r(T1X1, T2X2)
11:           add TX to agenda if new
```

We can then compute the invhom automaton I, such that $L(I) = h^{-1}(L(D_s))$. I uses the same states as D_s, but uses terminal symbols from Σ instead of Δ. Some rules of the invhom automaton I in the example are shown in Fig. 3b. Notice that I also contains rules that are not consistent with M, i.e. that would not occur in a grammatical parse of the sentence, such as $[2,4] \to \mathsf{r}_1([2,3],[3,4])$. Finally, the chart Ch_s is computed by intersecting M with I (see Fig. 3c). The states of Ch_s are pairs of states from M and states from I. It accepts τ_1, because $\tau_1 \in \mathsf{parses}(s)$. Observe the similarity to a traditional context-free parse chart.

3 Bottom-up intersection

Both the practical efficiency of this algorithm and its asymptotic complexity depend crucially on how we compute intersection and invhom. We illustrate this using an overly naive intersection algorithm as a strawman, and then analyze the problem to lay the foundations for the improved algorithms in Sections 4 and 5.

Let's say that we want to compute a tree automaton C for the intersection of a "left" automaton M_L and a "right" automaton M_R both over the same signature Σ. In the application to IRTG parsing, M_L is typically the derivation tree automaton (called M above) and M_R is the invhom of a decomposition automaton. As in the product construction for finite string automata, the states of C will be pairs TX of states T of M_L and states X of M_R, and the rules of C will all have the form $TX \to r(T_1X_1, \ldots, T_nX_n)$, where $T \to r(T_1, \ldots, T_n)$ is a rule in M_L, and $X \to r(X_1, \ldots, X_n)$ is a rule in M_R.

3.1 Naive intersection

A naive bottom-up algorithm is shown in Alg. 1.[1] This algorithm maintains an agenda of state pairs that have been discovered, but not explored as children of bottom-up rule applications; and a chart-like set P of all state pairs that have ever been popped off the agenda. The algorithm maintains the invariant that if TX is on the agenda or in P, then T and X are *partners* (written $T \approx X$), i.e. there is a tree $t \in T_\Sigma$ such that $T \to^* t$ in M_L and $X \to^* t$ in M_R.

The agenda is initialized with all state pairs TX, for which M_L has a rule $T \to r$ and M_R has a rule $X \to r$ for some nullary symbol $r \in \Sigma$. Then, while there are state pairs left on the agenda, Alg. 1 pops a state pair $T'X'$ off the agenda and adds it to P; iterates over all state pairs $T''X''$ in P; and queries M_L and M_R bottom-up for rules in which these states appear as children.[2] The iteration in line 7 allows T' and X' to be either left or right children in these rules. For each pair of left and right rules, the rules are combined into a rule of C, and the pair of the parent states T and X is added to the agenda.

This naive intersection algorithm yields an asymptotic complexity for IRTG parsing that is higher than expected. Assume, for example,

[1] We assume binary symbols for simplicity; all algorithms generalize to arbitrary arities.

[2] For the invhom automaton this can be done by substituting the variables in the homomorphic image $h(r)$ with the corresponding states X' and X'', and running the decomposition automaton on the resulting tree.

Algorithm 2 Bottom-up intersection with BU

```
1: initialize agenda with state pairs for constants
2: generate new S_r = S(M_R, r) for every r ∈ Σ
3: while agenda is not empty do
4:    T'X' ← pop(agenda)
5:    for T → r(T_1, T_2) in M_L s.t. T_i = T' do
6:       for X → r(X_1, X_2) ∈ BU(S_r, i, X') do
7:          store rule TX → r(T_1X_1, T_2X_2)
8:          add TX to agenda if new
```

that we are parsing with an IRTG encoding of a context-free grammar, i.e. with a string algebra (as in Fig. 1). Then the states of M_R are spans $[i, k]$, i.e. M_R has $O(n^2)$ states. Once line 4 has picked a span $X' = [i, j]$, line 6 iterates over all spans $X'' = [k, l]$ that have been discovered so far – including ones in which $j \neq k$ and $i \neq l$. Thus the bottom-up lookup in line 9 is executed $O(n^4)$ times, most of which will yield no rules. The overall runtime of Alg. 1 is therefore higher than the asymptotic runtime of $O(n^3)$ expected for context-free parsing. Similar problems arise for other algebras; for instance, the runtime of Alg. 1 for TAG parsing is $O(n^8)$ rather than $O(n^6)$.

3.2 Indexing

In context-free parsing algorithms, such as CKY or Earley, this issue is addressed through appropriate index datastructures, which organize P such that the lookup in line 5 only returns state pairs where X'' is of the form $[j, k]$ or $[k, i]$. This reduces the runtime to cubic.

The idea of obtaining optimal asymptotic complexities in IRTG parsing through appropriate indexing was already mentioned from a theoretical perspective by Koller and Kuhlmann (2011). However, they assumed an optimal indexing data structure as given. In practice, indexing requires algebra-specific knowledge about X'': A CKY-style index only works if we assume that the states of the decomposition automaton are spans (this is not the case in other algebras), and that the only binary operation in the string algebra is $*$, which composes spans in a certain way. Furthermore, in IRTG parsing the rules of the invhom automaton do not directly correspond to algebra operations, but to *terms* of operations, which further complicates indexing.

In this paper, we incorporate indexing into the intersection algorithm through *sibling-finders*. A sibling-finder $S = S(M, r)$ for an automaton M and a label r in M's signature is a data structure that supports a single operation, $\mathsf{BU}(S, i, X')$. We require that a call to $\mathsf{BU}(S, i, X')$ returns the set of rules $X \to r(X_1, \ldots, X_n)$ of M such that X' is the i-th child state, and for every $j \neq i$, X_j must be a state for which we previously called $\mathsf{BU}(S, j, X_j)$. Thus a sibling-finder performs a bottom-up rule lookup, changing its state after each call by caching the state and position.

Assume that we have sibling-finders for M_R. Then we can modify the naive Alg. 1 to the closely related algorithm shown as Alg. 2. This algorithm maintains the same agenda as Alg. 1, but instead of iterating over all explored partner states $T''X''$, it first iterates over all rules in M_L that have T' as a child (line 5). In line 6, Alg. 2 then queries M_R sibling-finders – we maintain one for each rule label – for right rules with matching rule label and child positions. Note that because there is only one rule with label r in M_L, the sibling-finders implicitly keep track of the partners of T_2 we have seen so far. Thus they play the role of a more structured variant of P.

There are a number of ways in which sibling-finders can be implemented. First, they could simply maintain sets $\mathsf{chi}(S_r, i)$ where a call to $\mathsf{BU}(S_r, i, X')$ first adds X' to $\mathsf{chi}(S_r, i)$. The query can then iterate over the set $\mathsf{chi}(S_r, 3-i)$, to check for each state X'' in that set whether M_R actually contains a rule with terminal symbol r and children X' and X'' (in the right order). This essentially reimplements the behavior of Alg. 1, and comes with the same complexity issues.

Second, we could theoretically iterate over all rules of M_R to implement the sibling finders via a bottom-up index (e.g., a trie) that supports efficient BU queries. However, in IRTG parsing M_R is the invhom of a decomposition automaton. Because the decomposition automaton represents all the ways in which the input object can be built recursively out of smaller structures, including ones which will later be rejected by the grammar, such automata can be very large in practice. Thus we would like to work with a lazy representation of M_R and avoid iterating over all rules.

4 Efficient bottom-up lookup

Finally, we can exploit the fact that in IRTG parsing, M_R is the invhom of a decomposition automaton. Below, we first show how to define algebra-specific sibling-finders for decompo-

Algorithm 3 passUpwards(Y, π, i, r)

```
rules ← BU(S_{r,π}, i, Y)
if π = π'k ≠ ϵ then
    for X → f(X_1, ..., X_n) ∈ rules do
        passUpwards(X, π', k, r)
```

sition automata. Then we develop an algebra-independent way to generate invhom sibling-finders out of those for the decomposition automata. These can be plugged into Alg. 2 to achieve the expected parsing complexity.

4.1 Sibling-finders for decomposition automata

First, consider the special case of sibling-finders for a decomposition automaton D. The terminal symbols f of D are the operation symbols of an algebra. If we have information about the operations of this algebra, and how they operate on the states of D, a sibling-finder $S = S(D, f)$ can use indexing specific to the operation f to look up potential siblings, and only for them query D to answer $\mathsf{BU}(S, i, X)$

For instance, a sibling-finder for the '$*$' operation of the string algebra may store all states $[k, l]$ for $i = 1$ under the index l. Thus a lookup $\mathsf{BU}(S, 2, [l, m])$ can directly retrieve siblings from the l-bin, just as a traditional parse chart would. Spans which do not end at l are never considered. Different algebras require different index structures. For instance, sibling-finders for the string-wrapping operation in the TAG string algebra might retrieve all pairs of substrings $[k, l, m, o]$ that wrap around $[l, m]$ instead. Analogous data structures can be defined for the s-graph algebra.

4.2 Sibling-finders for invhom

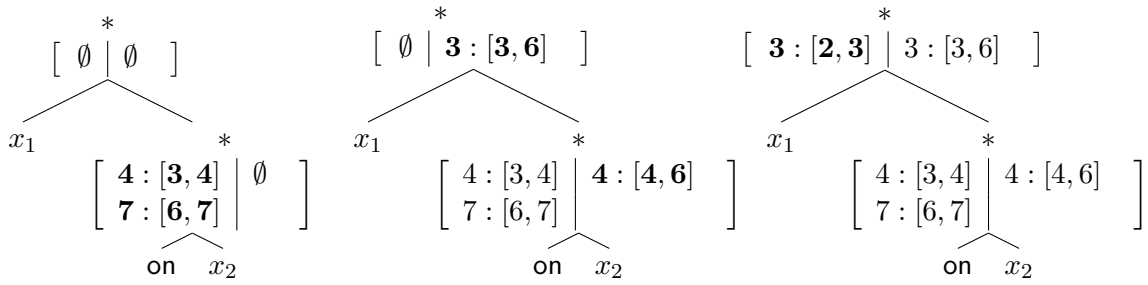
We can build upon the D-sibling-finders to construct sibling-finders for the invhom I of D. The basic idea is as follows. Consider the term $h(\mathsf{r}_1) = *(x_1, x_2)$ from Fig. 1. It contains a single operation symbol $*$ (plus variables); the homomorphism only replaces one symbol with another. Thus a sibling-finder $S(D, *)$ from the decomposition automaton can directly serve as a sibling-finder $S(I, \mathsf{r}_1)$. We only need to replace the $*$ label on the returned rules with r_1.

In general, the situation is more complicated, because $t = h(r)$ may be a complex term consisting of many algebra operations. In such a case, we construct a separate sibling-finder $S_{r,\pi} =$ new $S(D, t(\pi))$ for each node π with at least two children. For instance, consider the term $t = h(\mathsf{r}_4)$ in Fig. 1. It contains three nodes which are labeled by algebra operations, two of which are the concatenation. We decorate these with the sibling-finders $S_{\mathsf{r}_4,\epsilon}$ and $S_{\mathsf{r}_4,1}$. Each of these is a sibling-finder for the algebra's concatenation operation; but they may have different state because they received different queries.

We can then construct an invhom sibling-finder $S_r = S(I, r)$, which answers a query $\mathsf{BU}(S_r, i, X')$ in two steps. First, we substitute the variable x_i by the state X' and percolate it upward through t using the D-sibling-finders on the path from x_i to the root. If $\pi = \pi' k$ is the path to x_i, we do this by calling passUpwards(X', π', k, r), as defined in Alg. 3. If the local sibling-finder returns rules and we are not at the root yet, we recursively call passUpwards at the parent node π' with each parent state of these rules.

As we do this, we let each sibling-finder maintain the set of rules it found, indexed by their parent state. This allows us to perform the second step: we traverse t top-down from the root to extract the rules of the invhom automaton that answer the BU query. Recall that $\mathsf{BU}(S_r, i, X')$ should return only rules $X \to r(X_1, X_2)$ where $\mathsf{BU}(S_r, 3 - i, X_{3-i})$ was called before. Here, this is guaranteed by having distinct D-sibling-finders $S_{\pi,r}$ for every node π at every tree $h(r)$. A final detail is that before the first query to r, we initialize the sibling-finders by calling passUpwards for all the leaves that are labeled by constants.

This process is illustrated in Fig. 4, on the sibling-finder $S = S(I, \mathsf{r}_4)$ and the input string "John walks on a hill on Mars", parsed with a suitable extension of the IRTG in Fig. 1. The decomposition automaton can accept the word "on" from states $[3, 4]$ and $[6, 7]$, which during initialization are entered into position 1 of the lower D-sibling-finder $S_{\mathsf{r}_4,2}$, indexed by their end positions (a). Alg. 2 may then generate the query $\mathsf{BU}(S, 2, [4, 6])$. This enters $[4, 6]$ into the lower D-sibling-finder, and because there is a state with end position 4 on the left side of this sibling-finder, $\mathsf{BU}(S_{\mathsf{r}_4,2}, 2, [4, 6])$ returns a rule with parent $[3, 6]$. The parent is subsequently entered into the upper sibling-finder (b). Finally, the query $\mathsf{BU}(S, 1, [2, 3])$ enters $[2, 3]$ into the upper D-sibling-finder and discovers its sibling $[3, 6]$ (c). This yields a state $X = [2, 6]$ for the whole phrase



(a) After initialization. (b) After $\mathsf{BU}(S, 2, [4, 6])$. (c) After $\mathsf{BU}(S, 1, [2, 3])$.

Figure 4: Three stages of BU on $S(I, \mathsf{r}_4)$ for the sentence "John walks on a hill on Mars".

Algorithm 4 Top-down intersection

```
1: function expand(X):
2: if X ∉ visited then
3:     add X to visited
4:     for X → r(X1, X2) in M_R do
5:         call expand(X_i) for i = 1, 2
6:         for T→r(T1, T2) s.t. T_i ∈ prt(X_i) do
7:             store rule TX→r(T1X1, T2X2)
8:             add T to prt(X)
```

"walks on a hill". The top-down traversal of the sibling-finders reveals that this state is reached by combining $x_1 = [2, 3]$, for which this BU query asked, with $x_2 = [4, 6]$, and thus the BU query yields the rule $[2, 6] \to \mathsf{r}_4([2, 3], [4, 6])$. A subsequent query for $\mathsf{BU}(S, 2, [4, 8])$ would yield the rule $[2, 8] \to \mathsf{r}_4([2, 3], [4, 8])$, and so on.

The overall construction allows us to answer BU queries on invhom automata while making use of algebra-specific index structures. Given suitable index structures, the asymptotic complexity drops down to the expected levels, e.g. $O(n^3)$ for IRTGs using the string algebra, $O(n^6)$ for the TAG string algebra, and so on. This yields a practical algorithm that can be flexibly adapted to new algebras by implementing their sibling-finders.

5 Top-down intersection

Instead of investing into efficient bottom-up queries, we can also explore the use of top-down queries instead. These ask for all rules with parent state X and terminal symbol r. Such queries completely avoid the problem of finding siblings in M_R. An invhom automaton can answer top-down queries for r efficiently by running the decomposition automaton top-down on $h(r)$, collecting child states at the variable nodes. For instance, if we query I from Section 2 top-down for rules with the parent $[1, 5]$ and symbol r_1, it will enumerate the rules $[1, 5] \to \mathsf{r}_1([1, 2], [2, 5])$, $[1, 5] \to \mathsf{r}_1([1, 3], [3, 5])$, and $[1, 5] \to \mathsf{r}_1([1, 4], [4, 5])$, without ever considering any other combination of child states.

This is the idea underlying the intersection algorithm in Alg. 4. It recursively visits states X of M_R, collecting for each X a set $\mathsf{prt}(X)$ of states T of M_L such that $T \approx X$. Line 5 ensures that the prt sets have been computed for both child states of the rule $X \to r(X_1, X_2)$. Line 6 then does a bottom-up lookup of M_L rules with the terminal symbol r and with child states that are partners of X_1 and X_2. Applied to our running example, Alg. 4 parses "John walks on Mars" by recursive calls on $\mathrm{expand}([1, 5])$ and $\mathrm{expand}([2, 5])$, following the rules of I top-down. Recursive calls for $[2, 3]$ and $[4, 5]$ establish $\mathsf{VP} \in \mathsf{prt}([2, 3])$ and $\mathsf{NP} \in \mathsf{prt}([4, 5])$, which enables the recursive call for $[2, 5]$ to apply r_4 in line 6 and consequently add VP to $\mathsf{prt}([2, 5])$ in line 8.

The algorithm mixes top-down queries to M_R with bottom-up queries to M_L. Line 6 implements the core idea of the CKY parser, in that it performs bottom-up queries on sets of nonterminals that are partners of adjacent spans – but generalized to arbitrary IRTGs instead of just the string algebra. The top-down query to M_R in line 4 is bounded by the number of rules that actually exist in M_R, which is $O(n^3)$ for the string algebra, $O(n^6)$ in the TAG string algebra, and $O\left(n^s \cdot 3^{ds} ds\right)$ for graphs of degree d and treewidth $s - 1$ in the graph algebra. Thus Alg. 4 achieves the same asymptotic complexity as native parsing algorithms.

Condensed top-down intersection. One weakness of Alg. 4 is that it iterates over all rules $X \to r(X_1, X_2)$ of M_R individually. This can be extremely wasteful when M_R is the invhom of a decomposition automaton, because it may contain

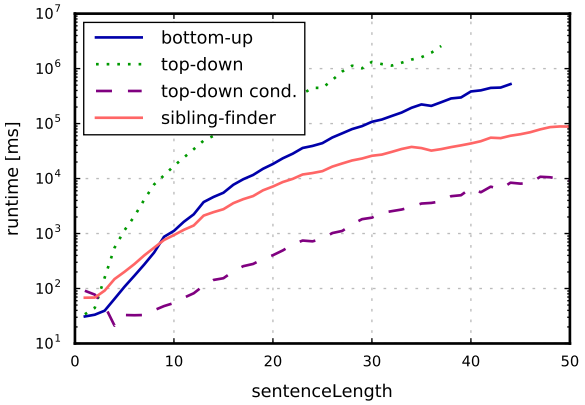


Figure 5: Runtimes for context-free parsing.

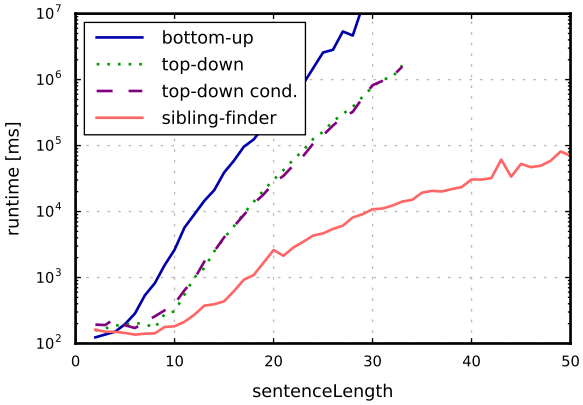


Figure 6: Runtimes for TAG parsing.

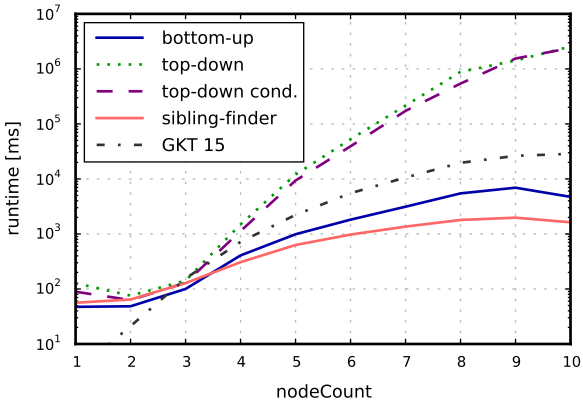


Figure 7: Runtimes for graph parsing.

a great number of rules that have the same states and only differ in the terminal symbol r. For instance, when we encode a context-free grammar as an IRTG, for every rule r of the form $\mathsf{A} \to \mathsf{B}\ \mathsf{C}$ we have $h(\mathsf{r}) = *(x_1, x_2)$. The rules of the invhom automaton are the same for all terminal symbols r with the same term $h(\mathsf{r})$. But Alg. 4 iterates over rules r and not over different terms $h(\mathsf{r})$, repeating the exact same computation for every binary rule of the context-free grammar.

To solve this, we define *condensed* tree automata, which have rules of the form $X \to \rho(X_1, \ldots, X_n)$, where $\rho \subseteq \Sigma$ is a nonempty set of symbols with arity n. A condensed automaton represents the tree automaton which for all condensed rules $X \to \rho(X_1, \ldots, X_n)$ and all $r \in \rho$ has the rule $X \to r(X_1, \ldots, X_n)$. It is straightforward to represent an invhom automaton as a condensed condensed automaton, by determining for each distinct homomorphic image t the set $\rho_t = \{r_1, \ldots, r_k\}$ of symbols with $h(r_i) = t$.

We can modify Alg. 4 to iterate over condensed rules in line 4, and to iterate in line 6 over the rules $T \to r(T_1, T_2)$ for which $T_i \in \mathsf{prt}(X_i)$ and $r \in \rho$. This bottom-up query to M_L can be answered efficiently from an appropriate index on the rules of M_L. Altogether, this condensed intersection algorithm can be dramatically faster than the original version, if the grammar contains many symbols with the same homomorphic image.

6 Evaluation

We compare the runtime performance of the proposed algorithms on practical grammars and inputs, from three very different grammar formalisms: context-free grammars, TAG, and HRG graph grammars. In each setting, we measure the runtime of four algorithms: the naive bottom-up baseline of Section 3; the sibling-finder algorithm from Section 4; and the non-condensed and the condensed version of the top-down algorithm from Section 5. The results are shown in Figures 5, 6 and 7. We measure the runtimes for computing the complete chart, and plot the geometric mean of runtimes for each input size on a log scale.

We measured all runtimes on an Intel Xeon E7-8857 CPU at 3 GHz using Java 8. The JVM was warmed up before the measurements. The parser filtered each grammar automatically, removing all rules whose homomorphic image contained a constant that could not be used for a given input (e.g., a word that did not occur in the sentence).

PCFG. We extracted a binarized context-free grammar with 6929 rules from Section 00 of the Penn Treebank, and parsed the sentences of Section 00 with it. The homorphism in the corresponding IRTG assigns every terminal symbol a constant or the term $*(x_1, x_2)$, as in Fig. 1. As a consequence, the condensed automaton optimization from Section 5 outperforms all other algo-

rithms, achieving a 100x speedup over the naive bottom-up algorithm when it was cancelled.

TAG. We also extracted a tree-adjoining grammar from Section 00 of the PTB as described by Chen and Vijay-Shanker (2000), converted it to an IRTG as described by Koller and Kuhlmann (2012), and binarized it, yielding an IRTG with 26652 rules. Each term $h(r)$ in this grammar represents an entire TAG elementary tree, which means the terms are much more complex than for the PCFG and there are much fewer terminal symbols with the same homomorphic term. As a consequence, condensing the invhom is much less helpful. However, the sibling-finder algorithm excels at maintaining state information within each elementary tree, yielding a 1000x speedup over the naive bottom-up algorithm when it was cancelled.

Graphs. Finally, we parsed a corpus of graphs instead of strings, using the 13681-rule graph grammar of Groschwitz et al. (2015) to parse the 1258 graphs with up to 10 nodes from the "Little Prince" AMR-Bank (Banarescu et al., 2013). The top-down algorithms are slow in this experiment, confirming Groschwitz et al.'s findings. Again, the sibling-finder algorithm outperforms all other algorithms. Note that Groschwitz et al.'s parser ("GKT 15" in Fig. 7) shares much code with our system. It uses the same decomposition automata, but a less mature version of the sibling-finder method which fully computes the invhom automaton. Our new system achieves a 9x speedup for parsing the whole corpus, compared to GKT 15.

7 Related Work

Describing parsing algorithms at a high level of abstraction has a long tradition in computational linguistics, e.g. in deductive parsing with parsing schemata (Shieber et al., 1995). A key challenge under this view is to index chart entries so they can be retrieved efficiently, which parallels the situation in automata intersection discussed here. Gómez-Rodríguez et al. (2009) present an algorithm that automatically establishes index structures that guarantee optimal asymptotic runtime, but also requires algebra-specific extensions for grammar formalisms that go beyond context-free string grammars.

Efficient parsing has also been studied in other generalized grammar formalisms beyond IRTG. Kanazawa (to appear) shows how the parsing problem of Abstract Categorial Grammars (de Groote, 2001) can be translated into Datalog, which enables the use of generic indexing strategies for Datalog to achieve optimal asymptotic complexity. Ranta (2004) discusses parsing for his Grammatical Framework formalism in terms of partial evaluation techniques from functional programming, which are related to the step-by-step evaluation of sibling-finders in Figure 4. Like the approach of Gomez-Rodriguez et al., these methods have not been evaluated for large-scale grammars and realistic evaluation data, which makes it hard to judge their relative practical merits.

Most work in the tree automata community has a theoretical slant, and there is less research on the efficient implementation of algorithms for tree automata than one would expect; Cleophas (2009) and Lengal et al. (2012) are notable exceptions. Even these tend to be motivated by applications such as specification and verification, where the tree automata are much smaller and much less ambiguous than in computational linguistics. This makes these systems hard to apply directly.

8 Conclusion

We have presented novel algorithms for computing the intersection and the inverse homomorphic image of finite tree automata. These can be used to implement a generic algorithm for IRTG parsing, and apply directly to any grammar formalism that can be represented as an IRTG. An evaluation on practical data from three different grammar formalisms shows consistent speed improvements of several orders of magnitude, and our graph parser has the fastest published runtimes.

A Java implementation of our algorithms is available as part of the Alto parser, `http://bitbucket.org/tclup/alto`.

We focused here purely on symbolic parsing, and on computing complete parse charts. In the presence of a probability model (e.g. for IRTG encodings of PCFGs), our algorithms could be made faster through the use of appropriate pruning techniques. It would also be interesting to combine the strengths of the condensed and sibling-finder algorithms for further efficiency gains.

Acknowledgments. We thank the anonymous reviewers for their comments. We are grateful to Johannes Gontrum for an early implementation of Alg. 4, and to Christoph Teichmann for many fruitful discussions. This work was supported by the DFG grant KO 2916/2-1.

References

Laura Banarescu, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer, and Nathan Schneider. 2013. Abstract meaning representation for sembanking. In *Proceedings of the Linguistic Annotation Workshop (LAW VII-ID)*.

John Chen and K. Vijay-Shanker. 2000. Automated extraction of TAGs from the Penn Treebank. In *Proceedings of IWPT*.

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

David Chiang. 2007. Hierarchical phrase-based translation. *Computational Linguistics*, 33(2):201–228.

Stephen Clark and James Curran. 2007. Wide-coverage efficient statistical parsing with CCG and log-linear models. *Computational Linguistics*, 33(4):493–552.

Loek Cleophas. 2009. Forest FIRE and FIRE Wood: Tools for tree automata and tree algorithms. In *Proceedings of the Conference on Finite-State Methods and Natural Language Processing (FSMNLP)*.

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

Philippe de Groote. 2001. Towards abstract categorial grammars. In *Proceedings of the 39th ACL/10th EACL*.

Michel Galley, Mark Hopkins, Kevin Knight, and Daniel Marcu. 2004. What's in a translation rule? In *Proceedings of HLT/NAACL*.

Carlos Gómez-Rodríguez, Jesús Vilares, and Miguel A. Alonso. 2009. A compiler for parsing schemata. *Software: Practice and Experience*, 39(5):441–470.

Jonathan Graehl, Kevin Knight, and Jonathan May. 2008. Training tree transducers. *Computational Linguistics*, 34(3).

Jonas Groschwitz, Alexander Koller, and Christoph Teichmann. 2015. Graph parsing with s-graph grammars. In *Proceedings of the 53rd ACL and 7th IJCNLP*.

Aravind K. Joshi and Yves Schabes. 1997. Tree-Adjoining Grammars. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3, pages 69–123. Springer-Verlag, Berlin.

Laura Kallmeyer and Wolfgang Maier. 2013. Data-driven parsing using probabilistic linear context-free rewriting systems. *Computational Linguistics*, 39(1):87–119.

Makoto Kanazawa. to appear. Parsing and generation as datalog query evaluation. *IfCoLog Journal of Logics and Their Applications*.

Alexander Koller and Marco Kuhlmann. 2011. A generalized view on parsing and translation. In *Proceedings of the 12th International Conference on Parsing Technologies (IWPT)*.

Alexander Koller and Marco Kuhlmann. 2012. Decomposing TAG algorithms using simple algebraizations. In *Proceedings of the 11th TAG+ Workshop*.

Ondrej Lengal, Jiri Simacek, and Tomas Vojnar. 2012. Vata: A library for efficient manipulation of non-deterministic tree automata. In C. Flanagan and B. König, editors, *Tools and Algorithms for the Construction and Analysis of Systems: 18th International Conference, TACAS 2012*. Springer.

Mike Lewis and Mark Steedman. 2014. A* CCG parsing with a supertag-factored model. In *Proceedings of EMNLP*.

Aarne Ranta. 2004. Grammatical framework: A type-theoretical grammar formalism. *Journal of Functional Programming*, 14(2):145–189.

Nina Seemann, Fabienne Braune, and Andreas Maletti. 2015. String-to-tree multi bottom-up tree transducers. In *Proceedings of the 53rd ACL and 7th IJCNLP*.

Stuart M Shieber, Yves Schabes, and Fernando CN Pereira. 1995. Principles and implementation of deductive parsing. *The Journal of logic programming*, 24(1):3–36.

Mark Steedman. 2001. *The Syntactic Process*. MIT Press, Cambridge, MA.

Yuk Wah Wong and Raymond J. Mooney. 2006. Learning for semantic parsing with statistical machine translation. In *Proceedings of the Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics (HLT/NAACL-2006)*.