Finite-state subset approximation of phrase structure

Mans Hulden and Miikka Silfverberg
University of Helsinki
{mans.hulden,miikka.silfverberg}@helsinki.fi

Abstract

We describe a method and a software tool to approximate and manipulate phrase structure grammars by a string representation of derivation trees and an encoding of a finite automaton that recognizes such strings. Many linguistically natural extensions to phrase structure grammars can be modeled on top of the approximation, allowing for a generic mechanism to model parsing and generation of a variety of phrase-structure based grammatical theories. Phenomena such as constituent movement, unification, and local constraints can be accommodated into the model by way of well-known mechanisms of finite-state manipulation. Additionally, the approximation supports the addition of probabilistic information to the syntactic structures encoded.

Introduction

Finite-state approximations of phrase structure grammars—context-free grammars (CFGs) in particular—are attractive and useful for various reasons. Deterministic finite state machines provide linear-time parsing in non-probabilistic scenarios and quadratic-time best-parse searches when performing probabilistic modeling with probabilistic context-free grammars (PCFGs). Finite-state approximations also open up possibilities of grammar engineering that would be unwieldy to express in a CFG formalism. For example, constraining an existing CFG with Boolean combinations of constraints on well-formedness is difficult to perform by manipulation of CFG rules. Equally, adding grammatical mechanisms to a simple phrase structure model—movement, coindexation constraints, long-distance constraints, unification, to name a few—requires complex modification to parsing and generation algorithms for dealing with computational models. By contrast, if a phrase-structure grammar is approximated to a reasonable degree as a finite-state device, such additional grammar manipulation becomes much less complicated by virtue of the closure properties of regular languages under Boolean operations.

In this paper we describe a technique for constructing a finite-state automaton (FSA) that accepts only descriptions of valid trees produced by a phrase-structure grammar, up to some predefined limit on the number of center-embeddings allowed. The automaton in question represents a subset approximation of the set of valid trees in the grammar. In other words, some valid trees that exhibit more than the desired number of center-embedded structures are not accepted by the automaton, while all the valid trees within the limit of approximation are distinguished. The particular encoding we use yields automata that are small enough to use in practice with large-scale grammars, and can incorporate a reasonable number of center-embeddings (up to 3 for large grammars induced from treebanks).

Once such automata are created, they not only serve as useful devices for efficient parsing of unembellished context-free grammars. In addition to this, one can take advantage of the flexibility and closure properties of the regular languages to express additional grammatical details, including:

- Additional constraints that depend on relationships between nodes of parse trees (for expressing binding relations, coindexation, etc.)
- Local grammatical constraints (lexical, head-specifier relations, etc.)
- Probabilistic information of nodes and lexical items (PCFGs)
- Unification models
- Constraints expressed in first-order logic

Many other grammar manipulation devices of the type that are encountered in the theoretical linguistics literature can also be incorporated.

Background

Previous approaches to regular approximation of phrase-structure grammars have focused largely on producing a finite-state machine (FSM) that *recognizes* strings accepted by some CFG, (Pereira and Wright 1991; Nederhof 2000; Mohri and Nederhof 2001)[1] or performing transforms on grammars to make it regular (Mohri and Nederhof 2001), or by a detour through alternative formalisms (Langendoen 2008). By contrast, in this work we instead develop a string encoding of the set of trees produced by a CFG and construct an automaton that recognizes a subset of the valid trees associated with a CFG. This string encoding can approximate a CFG up to some desired level of center-embedding.

[1](Nederhof 2000) provides a good overview of the dominant methods.

From our encoding, it is also possible to extract, by a simple homomorphism, only the set of terminal strings accepted by the grammar, discarding the tree structure if one so desires, thus producing an approximate finite-state acceptor instead of a parser. The automaton approximation in question can also deal with weights or probabilities in the grammar and can be used to model a probabilistic context-free grammar as well. The automaton produced by the approximation directly yields a linear time parsing method for all sentences within the purview of the approximation. Parsing is accomplished by intersecting an auxiliary automaton produced from a sentence to be parsed and the automaton that approximates the grammar. Similarly, one can also calculate the most probable parse given a PCFG approximation in $O(n^2)$-time. The grammar can also be reduced to a language model—a simple weighted automaton that can be used to calculate the probability of the most likely parse of a sentence given the PCFG in question without actually retrieving a parse.

The approximation strategy presented here is based on an intuition present—either explicitly or implicitly—in much of the earlier work attempting to improve the parsing efficiency of natural language phrase structure grammars. The intuition is that natural language sentences, when represented as phrase-structure trees, have a tendency to consist of long sequences of left or right-recursive substructures without center-embedding, as seen in figure 1. This observation has been used to approximate phrase structure grammars and CFGs in various ways. Langendoen and Langsam (1984; 1987) build a finite-state transducer that attempts to assign tree representations to input strings with a method that assumes that trees are largely of this structure. Similarly, (Johnson 1998) performs a left-corner transform on CFGs before approximating them, so as to be able to approximate more tree structure with smaller automata.

The particular way in which the trees are encoded in the current work takes advantage of this lack of embedding and produces small automata that exactly model both left and right recursive trees. More complex trees that exhibit multiple levels of center-embedding are produced by recursive substitution of these left and right-recursive trees at certain positions in the tree, the number of substitutions depending on the desired level of center-embedding that is to be accepted.

The paper is structured as follows. First, a string encoding for phrase structure trees is presented, and the method for constructing an automaton that accepts such strings is given. We then show how the same encoding can be used to represent trees produced by PCFGs. Subsequently, we discuss some properties of the encoding. Some implementation details are given, and the results of practical experiments with actual grammars are reported. We also examine the growth rate of the size of the automata that are produced by the method from grammars induced from the Penn Treebank. Additionally, we outline and discuss some possibilities for extensions and modifications of the approach.

Notation

In the following, we assume that we are given descriptions of arbitrary context-free grammars: a context-free grammar G is a 4-tuple (N, T, P, S), where N is the set of non-terminal symbols, T a set of terminal symbols, used in a set of production rules P of the form $A \rightarrow \beta$ where $A \in N$ and β is a sequence drawn from $(N \cup T \cup \epsilon)^+$. S is a symbol from N designated as the start symbol.

Additionally, we use extended regular expressions to model regular languages and relations and assume familiarity with the notational devices of A^*, $(A \cap B)$, $(A \cup B)$, AB, denoting the Kleene closure of A, intersection, union and concatenation of the regular languages A and B, respectively. We denote by $A \circ B$ the composition of transducers A and B. We also denote creating a repeating or identity transducer from a finite automaton or regular language by the operation $Id(A)$, as well as calculating the output projection of a transducer by range(T), producing an automaton or equivalently, a regular language.

Overview of method

The fundamental idea behind the approach is as follows: we first define a set of trees that represent specific fragments of the trees generated by a grammar, which we call LR-trees. These LR-trees consist of left recursive elements only to the left of the root, and right recursion only to the right of the root. In other words, we provide an exact representation of all trees that are pure left-branching to the left of the root, and pure right-branching on all branches to the right of the root. This gives us a tree shaped like the one pictured in figure 1.

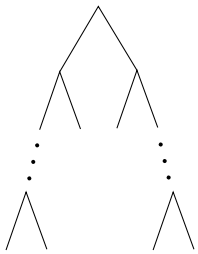


Figure 1: An LR-tree: a pure left and right branching tree.

As we shall show, we can construct a finite automaton that accepts string descriptions of all such structure fragments generated by some grammar, including ones that have arbitrarily deep left or right recursive substructure. Naturally, apart from such structures, the original grammar may also encode trees that cannot be expressed by this type of tree structure—that is, trees that have center-embedded structure, such as the tree in figure 2, if the new subtree headed by **X** were to be added to the initial tree. From an automaton that accepts only LR-trees constrained to be headed by the start symbol S, we create better approximations of the trees produced by the grammar by recursively substituting a finite number of times new LR-trees at the leaves in the string/automaton representation whenever the leaves of the

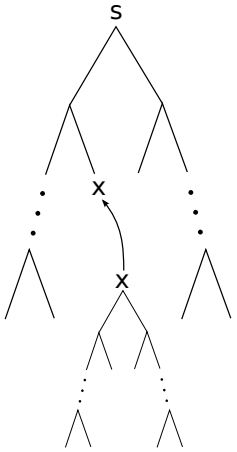


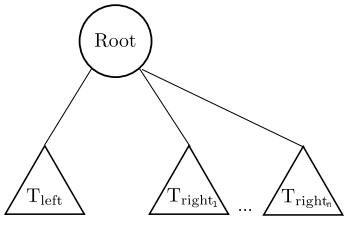
Figure 2: A left-right recursive tree headed by category S, the initial symbol in the CFG. The figure shows a possible addition of another tree at the site headed by category X, yielding a nesting depth 1.

tree are nonterminals. In practice, this is done by repeatedly modifying, at leaf positions, the automaton that accepts the initial tree. This modification can be modeled as a finite transducer, which can be composed repeatedly with the LR-tree description automaton, yielding deeper approximation of the grammar.

String representation

Any LR-tree that captures all pure left and right branching structures produced by a grammar can be represented as a string by traversing and listing the nodes in a certain specified order, given below. With such an encoding, the language representing the set of all such trees produced by a context-free grammar can also be encoded as a finite automaton. The automaton in question, which accepts the set of these limited trees represented as strings, can later be recursively substituted for at leaf positions, yielding new automata that accept trees encoding the set of valid trees in a CFG at some depth of center-embedding.

The linear string representation of a left-right-recursive tree is as follows. Given a tree T with a Root S, a left subtree $\mathrm{T}_{\mathrm{left}}$ and right subtrees $\mathrm{T}_{\mathrm{right}_1} \ldots \mathrm{T}_{\mathrm{right}_n}$, we:



(i) list the nodes of $\mathrm{T}_{\mathrm{left}}$ in postorder, mark nodes X along the spine with brackets (**[X]**), and mark non-spine nodes with braces ({ **X** }).

(ii) list the Root node S, and mark it by parentheses: **(S)**

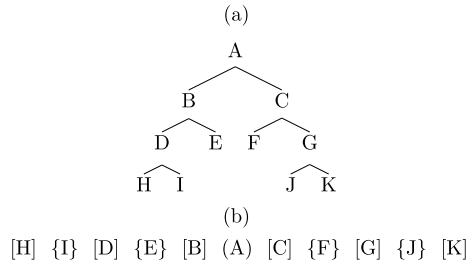


Figure 3: An example initial tree (a) and its postorder-root-preorder string representation (b).

(iii) list the nodes of $\mathrm{T}_{\mathrm{right}_1} \ldots \mathrm{T}_{\mathrm{right}_n}$ in preorder, mark nodes as in (i). Prepend the special symbol ˆ to each first node listed in $\mathrm{T}_{\mathrm{right}_n}$, if $n \geq 2$.

For example, in figure 3 we see an initial (in this case binary) tree, together with its string representation as described above. We make no distinction here between terminals and nonterminals: in the figure, the symbols **E,F,I,J** may be terminals or nonterminals. In the latter case, the tree would represent an 'unfinished' tree which needs to be expanded in the leaf positions by center-embedding.

Extending this notation to arbitrary trees requires us to apply the above definition recursively. For the recursive case, we assume that only nodes that do not have descendants (nodes that do not occur along the spine of the original tree) can recurse, and that any recursion is marked off by special symbols $< \ldots >$. Figure 4 shows a tree and its string representation using one level of recursion.

Focusing on a string representation of a context-free parse tree allows us to abstract away from the states and transitions when constructing the automaton that encodes an approximation of the valid trees in the grammar. Instead, we can focus on the nature of the strings that encode the tree, and express the automaton or regular language as a collection of constraints on the form of these strings.

Constructing an automaton for the string representation

Let $\Sigma_{T \cup N}$ be a joint alphabet of terminals and nonterminals in some context-free grammar $\mathcal{G}$, and define an auxiliary alphabet $\Sigma_{aux} = \{(,), \{, \}, [,], <, >, \hat{}\}$. Trees produced by the grammar $\mathcal{G}$ are represented as strings of the format:

$$\Big([\, \Sigma_{T \cup N}] \;\cup \{\, \Sigma_{T \cup N}\} \;\cup (\, \Sigma_{T \cup N}) \;\cup \hat{} \Big)^*$$

That is, they consist of a sequence of terminals and nonterminals, each surrounded by braces, parentheses, or brackets, as in figure 3. Additionally, we want to constrain the marked strings in such a way that they correspond only to valid complete or partial derivations produced by some context-free grammar $\mathcal{G}$. In other words, a parenthesized nonterminal should only occur once in a string, representing the root of the tree; nonterminals or terminals in braces should only occur at leaf positions, bracketed nonterminals

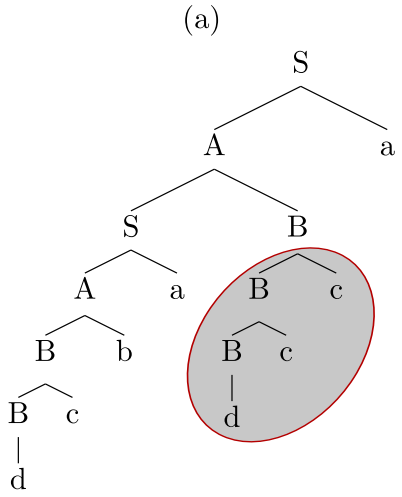


Figure 4: A tree (a) and its string representation (b). Parts of the string marked in bold indicate sections of one level of nesting in the encoding.

or terminals should only occur at spine positions, and the ^-separator should only occur before the third and subsequent right-branching branches.

Local constraints on well-formedness

To construct a finite automaton $\mathcal{LR}$ that accepts all and only those pure left and right-branching trees that stem from some context-free grammar $\mathcal{G}$, we need to construct a FSM over the alphabet $\Sigma = \Sigma_{T \cup N} \cup \Sigma_{aux}$, where each word fulfills the following conditions.

1. For each symbol position in a string to the left of the root (which is marked by parentheses), the substring $[A]\{B_1\} \ldots \{B_n\}[C]$ shall be accepted iff $\mathcal{G}$ contains a rule $C \rightarrow A\ B_1\ \ldots\ B_n$. All B_i may be empty.
2. For each symbol position in a string to the right of the root, the substring $[A]\{B_1\} \ldots \{B_n\}[C]$ shall be accepted iff $\mathcal{G}$ contains a rule $A\ \rightarrow\ B_1\ \ldots\ B_n\ C$. All B_i may be empty.
3. A substring $[A](B)[C_1]\ \ldots\ \hat{}\,[C_2]\ \ldots\ \hat{}\,[C_n]$ is accepted iff $\mathcal{G}$ contains a rule $B \rightarrow A\ C_1 \ldots C_n$ (where all C_i may be empty)

This automaton, $\mathcal{G}$, which accepts all the descriptions of trees produced by a context-free grammar headed by some root category X, is a regular language.[2] It is a fairly straightforward task to produce an automaton $\mathcal{LR}$ that rejects all strings that do not conform to the requirements given above, and accepts those that do. This can be done by constructing

[2]In fact, it is subregular: the language is k-testable.

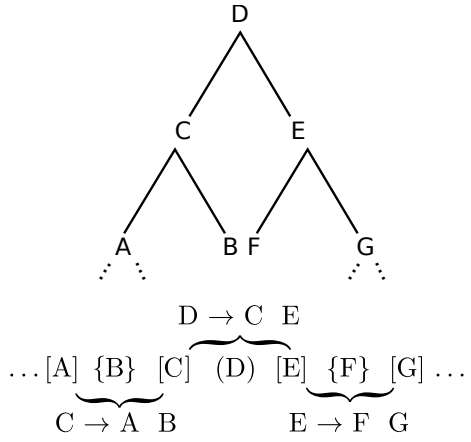


Figure 5: Example tree and string fragment illustrating the construction of the local constraints that together model the valid phrase structure trees in string format.

individual automata that check for the conditions 1–3 and combining them with Boolean operations.

Having such an automaton $\mathcal{LR}$ at our disposal, we can also create an initial tree $\mathcal{I}$ (which represents only those structures that are headed by the start symbol S) by intersecting $\mathcal{LR}$ with the language that contains one instance of a parenthesized start symbol S:

$$\mathcal{I} = \mathcal{LR} \cap \Sigma^* \ (\ S\ )\ \Sigma^* \tag{1}$$

That is, the automaton representing the initial tree, $\mathcal{I}$, accepts only those left and right branching trees that are rooted. Given this initial tree $\mathcal{I}$, we can proceed with deeper approximation. This is performed by repeatedly substituting, in appropriate positions in that automaton $\mathcal{I}$, new instances of $\mathcal{LR}$ at the transitions corresponding to leaf symbols, provided the leaves are nonterminals. This substitution is conveniently done by defining a finite-state transducer (FST) that replaces instances of strings

$$\{\ X\ \}$$

with strings representing the tree headed by the corresponding to the leaf non-terminal X. This can be calculated as:

$$\mathcal{LR} \cap \Sigma^* \ (\ X\ )\ \Sigma^*$$

The transducer also surrounds the replacement site with the recursion markers $< \ldots >$. Such a replacement is shown in figure 4. We define a generic transducer T_{Insert} that performs such replacements, and can now perform the recursive substitution of the leaves as repeated composition of transducers and extraction of the output projection (range) of the result:

$$\text{range}(Id(\mathcal{I}) \circ T_{\text{Insert}} \circ \ldots \circ T_{\text{Insert}})$$

where $Id(\mathcal{I})$ is the identity relation from the automaton $\mathcal{I}$.

After this process of repeated "insertion" of trees in the replacement sites is performed to the desired level of approximation, we need to additionally filter out strings that do not contain terminal symbols in all the leaf positions. These are 'unfinished' trees that could only generate complete terminal strings after further iterations. To this end, we construct a simple filter automaton F_l that rejects all words containing string sequences { **X** }, where **X** is a nonterminal. This intersection of the filter automaton and the grammar automaton yields the final approximation:

$$\mathcal{A}_{CFG} = \text{range}(Id(\mathcal{I}) \circ T_{\text{Insert}} \circ \ldots \circ T_{\text{Insert}}) \cap F_l \quad (2)$$

In what follows, we refer to the language defined by the initial tree $\mathcal{I}$ as level 0-nesting, and the result of subsequent expansions of the language range($\mathcal{I} \circ T_{\text{Insert}}$) as level 1-nesting, range($\mathcal{I} \circ T_{\text{Insert}} \circ T_{\text{Insert}}$), level 2-nesting, etc. See figure 6 for an illustration.

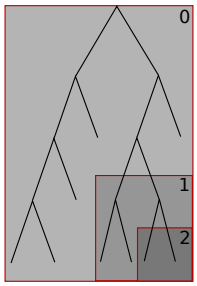


Figure 6: Nesting levels and order of approximation illustrated.

Probabilistic grammars

With the above method, we can also encode trees produced by derivations of PCFGs. This is done by first building a regular automaton approximation up to the levels of nesting desired, and then adding weights to certain transitions in the automaton. In particular, for each production rule observed in the tree, which is always represented in the format (X) or [X] in the string encoding, we add the weight (or probability) of the corresponding rule to the transition representing X, yielding an approximation $\mathcal{LR}_{PCFG}$.

Properties of the encoding

There are two noteworthy aspects of the particular encoding of trees as strings that we have chosen. First, the linear order of the terminals is maintained. This makes parsing a relatively straightforward task, given that we have an automaton that encodes the set of valid trees to some desired depth: we can simply construct a "sentence automaton" $\mathcal{A}_{SENTENCE}$ from the sentence to be parsed; this automaton accepts the sequence of terminals in question, with arbitrary nonterminals and brackets possibly interspersed, and then intersect this automaton with the grammar automaton. In other words, to parse a string, we calculate:

$$\mathcal{A}_{PARSE} = \mathcal{A}_{CFG} \cap \mathcal{A}_{SENTENCE} \quad (3)$$

S → A a
A → S B
A → B b
B → B c
B → d

Figure 7: A strongly regular grammar.

which yields an automaton that contains all the valid parses in the string representation.

A transducer for parsing

We may note in passing that the above method for parsing a sentence can also be used to create a finite transducer $\mathcal{T}_{PARSE}$ that, given an input sentence of terminal strings, directly parses it and outputs the corresponding structure. This is done by the standard technique of creating a transducer that inserts arbitrary sequences of nonterminals and auxiliary symbols into an input string and composing this with the identity transducer that represents the approximation:

$$\mathcal{T}_{PARSE} = (\epsilon \times (\Sigma_{aux} \cup \Sigma_N) \cup Id(\Sigma_T))^* \circ Id(\mathcal{A}_{CFG}) \quad (4)$$

Structural complexity

In contrast to many other possible approximation schemes of trees that grow exponentially with the lengths of sentences,[3] the size of the current automaton approximation grows exponentially only with the number of left or right turns needed in tree traversal, not with the length or other type of complexity of the sentence. This coincides with observations about complexity in human parsing of natural language structure, which has been associated with the number of zig-zag moves necessary when traversing a sentence tree top-down (Langendoen 2008).

Also, the fact that the automaton that represents the left and right recursive trees places only local restrictions on the well-formedness of the string makes it very compact. The decision to accept or reject a substring of $\mathcal{LR}$ can be made in all cases by consulting the previous or following n terminal or nonterminal symbols (for a rule of length n) in conjunction with the symbol currently being read.

Another property of the construction method is that if the CFG $\mathcal{G}$ is not self-embedding, the resulting regular grammar can be made exact, given enough substitutions. This follows from the fact that a non self-embedding grammar can only make a finite number of left-recursive or right-recursive turns, traversing the tree from the root downwards. For example, the tree in figure 4 was produced by the grammar in figure 7, an example context-free grammar that is in fact strongly regular (Nederhof 2000). When compiled into an automaton with the method described, the grammar becomes exact after two iterations of the substitution operation

[3] As does for instance any method of approximation that encodes trees as bracketed strings in the standard way. Although sometimes even this method can be useful if handled in the right way (Roche 1997).

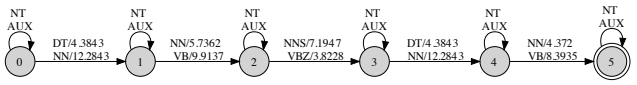


Figure 8: Example sequence of weighted preterminals to be intersected with grammar automaton for parse.

(composition with T_{Insert}, i.e. nesting level 2), and the final automaton that accepts all and only the valid trees generated by the grammar consists of 31 states. That it is exact can be ascertained by verifying that the automaton representing the approximation does not change on subsequent iterations of the substitution operation.

Implementation

We have implemented the method described above in an application that accepts as input an arbitrary context-free grammar (CFG/PCFG), and produces an automaton approximation up to the desired level of nesting capability. In practice, the tool reads (P)CFGs as text files, and produces a (possibly weighted) automaton in the AT&T FSM interchange format. The tool also contains utilities for parsing strings once such automata have been built as well as for extracting CFG recognizers from the tree approximations.

The main CFG to automaton conversion was implemented with the *foma* finite-state toolkit (Hulden 2009). In particular, the constraints on the well-formedness of the strings representing the tree were encoded in the mixture of regular expression and first-order logic formalism that *foma* provides. Subsequently, for probabilistic grammars, we weighted the transitions according to the PCFG using the Helsinki Finite State Toolkit (HFST) and API (Lindén, Silfverberg, and Pirinen 2009), which uses as its back-end the OpenFST toolkit (Allauzen et al. 2007). Parsing is performed by adding arbitrary nonterminal and auxiliary self-looping transitions to each state in an automaton that accepts the input sentence (see figure 8), and intersecting the automaton with the grammar automaton using *HFST*.

The extraction of the most probable parse, or a list of the n-best parses, makes use of the algorithm included in *HFST* (Mohri and Riley 2002). In the event that only a language model is desired and we only need to assign probabilities to word sequences, we first construct the approximation, and then substitute all non-terminal and auxiliary symbol transitions with ϵ-transitions (retaining the weights), and then perform an ϵ-removal operation, followed by possible determinization and minimization (Mohri and Riley 2002). For calculating with a language model, we can either operate with the tropical semiring (which would model a Viterbi assumption) and only return the probability of the most likely parse, in which case we produce smaller automata, or, to produce the sum of the probabilities of all parses, we operate with the log-semiring when performing the ϵ-removal.[4]

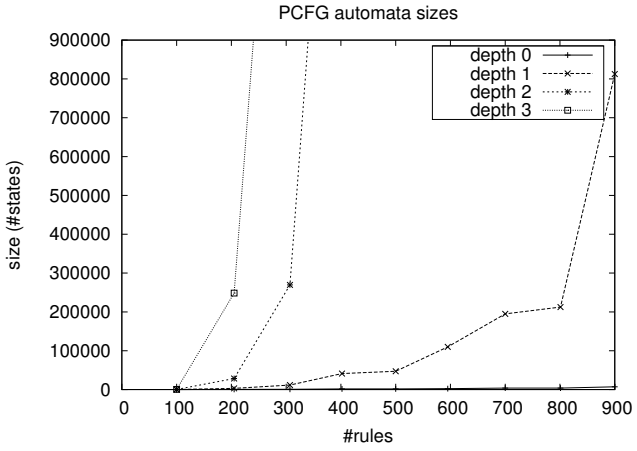


Figure 9: Growth in automaton sizes with PCFG approximation of Penn Treebank rules.

Practical experiments

To test the behavior of the encoding under the strain of actual grammars we extracted various subsets of grammars induced from the Wall Street Journal portion of the Penn Treebank (PTB) (Marcus, Marcinkiewicz, and Santorini 1993) and converted them into automata with varying degrees of approximation. Because of the special nature of the PTB—trees are quite flat and punctuation requires embedding—which tends to hurt approximation efforts, we made some additional changes to the above method and the grammars.

- Terminals (words) were not included in the approximation: rather, the grammars were constructed up to the preterminals—this is a special class of nonterminal that is always followed by a terminal in the PTB notation. Doing so does not hurt parsing, since at parsing time, the word/preterminal combinations can be looked up separately, and a sentence automaton (with appropriate weights) can be built; this is then intersected with the grammar automaton. This maneuver avoids having to store all the terminals used in the grammar (the individual words in the treebank).
- Chains of nonbranching nonterminals were included in each level of nesting. That is, $NT \rightarrow NT \rightarrow \ldots \rightarrow NT \rightarrow T$-chains were added to leaves where they would otherwise fall outside the nesting level (for example, in the *WHNP-1-WDT* sequence in the tree in figure 10, the node *WDT* would fall into nesting level 1 unless this addition was made). Since this final tree expansion is strictly local in nature, it causes no real growth in automaton complexity.
- Self-embedding punctuations were removed, i.e. rules like $S \rightarrow NP\ VP\ .$ were modified to disregard punctuation, as they put a burden on the approximation because of the nesting.

[4] Although the log-semiring is difficult to use in practice because of its very slow compilation time.

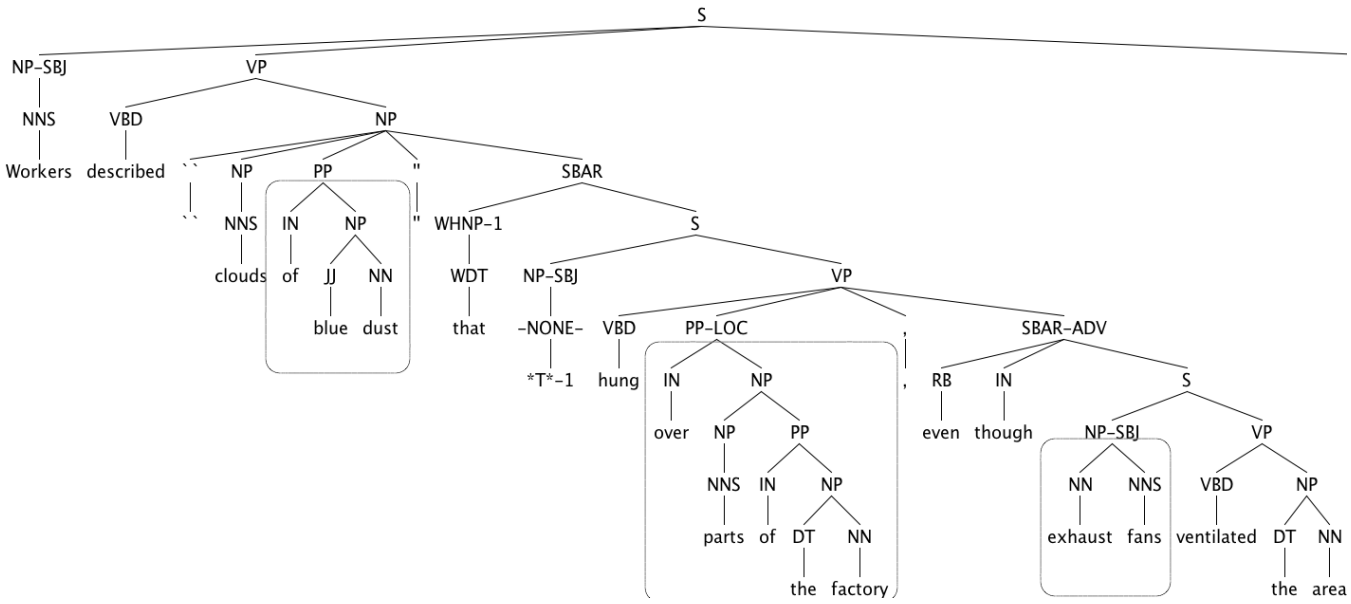


Figure 10: A moderately complex tree with those sections that reach nesting depth 1 marked.

- Given evidence of several rules of type $X \rightarrow Y^i Z$ for various i, such rules were collapsed into $X \rightarrow Y^* Z$, which again limits complexity in the automaton that encodes the approximation.

The growth of the sizes of the automata is given in figure 9. As is seen, to be able to handle large grammars, we need to restrict ourselves to nesting level 1 (with the above caveats and extensions). However, grammars of up to 500 rules are approximable in practice with nesting levels 2 or 3. With nesting level 1, grammars of several thousand rules can be approximated. Given the above modifications, very rarely are nesting levels 2 and beyond needed for accurate parsing. This is illustrated in figure 10, where the boxed sections show nesting level 1 subtrees. Should these subtrees still exhibit self-embedding, the approximation would need to move beyond level 1 to capture them.

Further work

The current encoding and method open up possibilities for grammar engineering that is either difficult or complex when handling (P)CFGs in more traditional ways. The fact that we obtain a regular language that accepts a string representation of a linguistically motivated subset of the valid trees that a grammar generates, allows us to further manipulate the set of trees in flexible ways.

Adding local constraints

One straightforward modification of a grammar $\mathcal{G}$ is to introduce local constraints on the structure of trees. For example, figure 11 illustrates in (a) a tree fragment where it would be desirable to express a dependency or constraint between two elements. If the CFG were to be directly modified to capture a constraint stating that, for example, X and Z may not co-occur, this would require a modification of several rules in the grammar. However, using our string representation, such local constraints can easily be added by intersecting the FSM $\mathcal{A}_{CFG}$ that represents the subset approximation of the grammar with another FSM that rejects string representations of the disallowed configuration; in this case we would create an automaton F that rejects words that contain substrings {**X**}**[Y]**{**Z**}**[W]**, and calculate a new approximation $\mathcal{A}'_{CFG} = (\mathcal{A}_{CFG} \cap F)$.

Movement and transformations

In a similar fashion, we may add modeling of movement and transformation operations to the approximation. Movement of elements in a tree can be added after constructing the initial phrase-structure approximation by appropriate combinations of transducer and automaton manipulation. Consider a CFG that generates some element X in the base position low in the tree, as in figure 11 (b). Now, if the actual surface form desired is one that results from moving X to some other position in the tree, again, this would be difficult to capture by simple changes to the CFG rules. By contrast, in the approximation representation, we can design a finite-state transducer that moves an element in the string representation, compose the original grammar automaton with the movement-transducer $\mathcal{M}$, and extract the range of this composition operation, producing a new automaton $\mathcal{A}'_{CFG}$ that accepts trees as they appear post-movement:

$$\mathcal{A}'_{CFG} = \text{range}(Id(\mathcal{A}_{CFG}) \circ \mathcal{M})$$

Other enhancements

Many other enhancements to the approximation strategy are also possible. For instance, unification-like co-occurrence restrictions are a natural candidate in this respect, as they

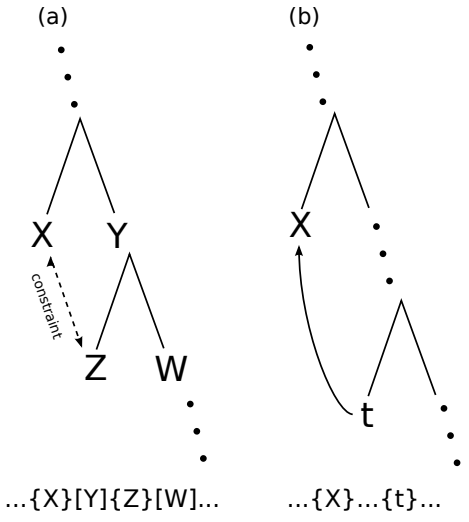


Figure 11: Examples of (a) constraints and (b) movement, both illustrated together with the tree and a fragment of the string representation.

could probably be compactly encoded into the approximation. This is because the preorder-postorder traversal of the tree nodes in the encoding would often tend to bring elements to be unified (in real linguistic grammars) very close to each other in the string representation, a configuration suitable for efficient approximation with automata.

Conclusion

We have presented a method of context-free grammar approximation and parsing through a regular subset approximation of arbitrary context-free grammars.

Experiments on converting large-scale grammars to automata using the approach indicate that it is feasible for accurately modeling context-free grammars with up to a few thousand rules and up to two levels of center embedding.

Experiments on re-parsing the Penn Treebank using an automaton of second-level embedding indicate that a number of complex sentences still fall outside the capabilities of first-level embedding. However, there are a number of ways in which the current subset approximation could be extended to capture these borderline cases, or perhaps all attested sentences in such a treebank, without much further growth in the state complexity. For example, one could selectively expand certain nodes (and not all) beyond nesting levels 0 and 1, perhaps depending on their frequency in embedded constructions.

The method is usable in practice: we have developed software tools to convert (P)CFGs to automata and parse sentences using the approximation. The approach offers several potential improvements and enhancements that are linguistically interesting.

References

Allauzen, C.; Riley, M.; Schalkwyk, J.; Skut, W.; and Mohri, M. 2007. OpenFST: A general and efficient weighted finite-state transducer library. In *Proceedings of the 12th international conference on Implementation and application of automata*, 11–23. Springer.

Hulden, M. 2009. Foma: a finite-state compiler and library. In *Proceedings of the 12th Conference of the European Chapter of the Association for Computational Linguistics*, 29–32. Association for Computational Linguistics.

Hulden, M. 2011. Parsing CFGs and PCFGs with a Chomsky-Schützenberger representation. In *Human Language Technology. Challenges for Computer Science and Linguistics*. Springer. 151–160.

Johnson, M. 1998. Finite-state approximation of constraint-based grammars using left-corner grammar transforms. In *Proceedings of the 17th international conference on Computational linguistics*, 619–623. Association for Computational Linguistics.

Langendoen, D. T., and Langsam, Y. 1984. The representation of constituent structures for finite-state parsing. In *Proceedings of the 22nd annual meeting on Association for Computational Linguistics*, 24–27. Association for Computational Linguistics.

Langendoen, D. T., and Langsam, Y. 1987. On the design of finite transducers for parsing phrase-structure languages. *Mathematics of Language* 191–235.

Langendoen, D. T. 2008. Coordinate grammar. *Language* 84:691–709.

Lindén, K.; Silfverberg, M.; and Pirinen, T. 2009. HFST tools for morphology–an efficient open-source package for construction of morphological analyzers. *State of the Art in Computational Morphology* 28–47.

Marcus, M.; Marcinkiewicz, M.; and Santorini, B. 1993. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics* 19(2):313–330.

Mohri, M., and Nederhof, M. 2001. Regular approximation of context-free grammars through transformation. In Junqua, J., and van Noord, G., eds., *Robustness in Language and Speech Technology*. Kluwer. 153–163.

Mohri, M., and Riley, M. 2002. An efficient algorithm for the n-best-strings problem. In *Seventh International Conference on Spoken Language Processing (ICSLP 02)*.

Nederhof, M. 2000. Practical experiments with regular approximation of context-free languages. *Computational Linguistics* 26(1):17–44.

Oflazer, K. 2003. Dependency parsing with an extended finite-state approach. *Computational Linguistics* 29(4):515–544.

Pereira, F., and Wright, R. 1991. Finite-state approximation of phrase structure grammars. In *Proceedings of the 29th annual meeting on Association for Computational Linguistics*, 246–255. Association for Computational Linguistics.

Roche, E. 1997. Parsing with finite-state transducers. In Roche, E., and Schabes, Y., eds., *Finite-state Language Processing*, 241–282. MIT Press.

Yli-Jyrä, A. 2003. Regular approximations through labeled bracketing. In *Proc. FGVienna, The 8th conference on Formal Grammar, Vienna, Austria*.