A Fast Filtering Algorithm for Massive Context-free Grammars

Jeremy Dohmann and Kyle Deeds
Harvard College
Cambridge, MA, USA
{dohmann,kdeeds}@college.harvard.edu

Abstract

All non-statistical context-free parsers are plagued by a worst case parse time, $O(|G| \times n^3)$, linear in the size of the grammar, $|G|$, which in many applications can cause major slowdowns [6]. In this paper, we pursue general purpose pre-processing intended to speed up the run time of context-free parsers by selecting a smaller fragment of the grammar prior to parsing. Our work provides an ad-hoc method to perform subgrammar selection, increasing the practicality of parsing with very large context-free grammars. We present an algorithm we call 'Terminal-tree filtering' (TTF), a new variant of the 'b-filtering' algorithm presented in [6], which finds the same filtered set of grammar rules as the 'b-filter' for a given input, but which achieves new state-of-the-art run time and grammar size performance within in this problem space. Our work boasts remarkable performance boosts across a range of workloads, including a 10-20 fold speedup on a real world English grammar of size $|G| \approx 115,000,000$. Furthermore, we achieve far greater scale, parsing sentences using grammars ≈ 10 times larger than those previously studied. The TTF algorithm filters quickly enough to provide feasible parsing times on our example English grammar. On this very same test suite [6] 's algorithms, which are the broadest, most recent contributions to the field, demonstrate unacceptable run times. We also provide a theoretical analysis which elucidates under which conditions worst-case behavior and best-case behavior can be expected, and show that even under worst-case performance, variants of the TTF still show some empirical advantages to the b-filter.

CCS Concepts

• **Theory of computation → Grammars and context-free languages**.

Keywords

context-free grammars, parsing, algorithms, filtering

ACM Reference Format:
Jeremy Dohmann and Kyle Deeds. 2020. A Fast Filtering Algorithm for Massive Context-free Grammars. In *2020 ACM Southeast Conference (ACMSE 2020), April 2–4, 2020, Tampa, FL, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3374135.3385266

ACMSE 2020, April 2–4, 2020, Tampa, FL, USA

ACM ISBN 978-1-4503-7105-6/20/03…$15.00
https://doi.org/10.1145/3374135.3385266

1 INTRODUCTION

1.1 Context-free Languages and Applications

Context-free languages are a formal class of language originally described by Noam Chomsky [9] and subsequently studied in great depth by linguists and computer scientists due to their usefulness in natural language processing and compilers. Parsing natural language is central to approaches in machine translation [3], natural language inference [8] , search engines [29], human computer interactions [23], and many other applications [15].

Though in recent years CFL parsers have ceded some of their supremacy to alternative grammar formalisms and syntactic representations such as dependency parsing [13] and combinatory-categorial grammars [12], many of the most prominent syntactic theories in linguistics are still rooted in context-free languages. This is primarily due to their intuitive form, broad generality, and interesting representational and computational properties which permit their application to problems far outside the world of NLP [22] [2] [25].

Among their formal properties, CFLs have been argued to be sufficiently expressive to represent nearly every syntactic phenomenon of interest in natural language [24] with limited exceptions [27].

1.2 Context-free Grammars and Scalability

In addition to their expressiveness and parsing guarantees, CFLs are made actionable by their capacity to be expressed as a context-free grammar (CFG), an intuitive formalism that describes the unbounded productions of a language as a finite rewrite system containing rules of the form:

$$A \rightarrow C \mid BC \mid ....$$

This rule would be read as "A expands to C, or B then C, or...".

An important result is that all CFGs can be reduced to Chomsky-normal form [10]. A grammar is said to be in Chomsky-normal form if all rules either contain two nonterminals, a single terminal or the empty string on the RHS.

The mapping between CFLs and CFGs is many to one. Additionally, in contrast to regular languages (a.k.a regular expressions, or finite state automata), the task of finding the *provably minimal* CFG that expresses a CFL is undecidable [17]. These two properties make the task of reducing grammar size without reducing expressiveness an open problem. Additionally, grammar reduction is of utmost importance in large-scale parsing applications because algorithms quickly become intractably slow and memory intensive as grammar sizes grow [6].

The main reasons for this intractability are that 1.) the run time of parsing algorithms is proportional to the grammar size and 2.) bottom-up dynamic programming algorithms, such as the Earley

algorithm waste a huge proportion of their memory pursuing dead ends during parse time whenever the number of grammar rules not applicable for the parsed input is much greater than the number of applicable rules - a property which, intuitively, should scale with grammar size in many applications.

1.3 Statistical Parsing

Avoiding the computational blow-up associated with processing the whole grammar for each parse is an urgent matter in the field of context-free grammars.

One of the most widespread ways of skirting this problem is to statistically annotate the grammar (e.g. using probabilistic context-free grammars [19] or neural networks [7]) so as to enable empirical heuristics which pursue a limited subset of potential incomplete parses at a time. A further potential limitation of statistical parsing is that it typically returns a fixed, finite number of results, pruned for maximum likelihood - while this design is crucial for avoiding the blowup associated with large grammars, it makes it so that statistical parsers don't return a representation containing *all* possible parses of a sentence, only *some*.

1.4 Non-statistical Parsing and Limitations

This paper concerns itself with enhancing non-statistical parsing techniques to skirt the computational blow-up associated with large grammars. Even though probabilistic grammars have achieved great successes, they are only useful in domains where large tagged corpora are available. There are many problem domains in which syntactic parsing is desirable but annotated data sets aren't available and the grammars needed to describe the problem are too large to be feasibly processed by the former state of the art filtering algorithms featured in [6]. Furthermore, as noted above, non-statistical parsing is the only true solution when the user desires *all* possible parses.

Experiment 3 presents one such use-case from an industry collaboration with Charles River Analytics, which achieved meaningful run time improvements on a domain-expert designed English grammar for use in the cybersecurity domain.

2 RELATED WORK

Within the non-statistical domain, there are two main methods by which to improve parse times and reduce the memory overhead associated with pursuing dead-ends in the grammar: grammar filtering [6] and guided parsing [4]. Guiding is a generalization of filtering, which was first suggested to the community in the pursuit of linear-time parsing [20].

Filtering is a pre-processing method in which, prior to parsing input s_i, an algorithm finds a subset G_{s_i} of the original grammar G such that all possible parses of s_i under G are admissible under G_{s_i}. This process can either proceed by strategies which preserve the expressiveness of the underlying CFL or by modifying the expressiveness of the CFL to be a sublanguage which generates s_i but potentially narrows its scope to omit some other elements of the original language.

Guiding is a somewhat more sophisticated, parsing-time method, in which, at every step of the parsing algorithm during which nondeterminism could be introduced, instead of exploring all possible rules at once, use some procedure (called an 'oracle' in [4]) to eliminate any rules which certainly cannot be applied.

Filtering is a degenerate form of guiding, where the same nondeterminism reduction criterion is applied at every step: don't expand a rule in G unless it's also in G_{s_i}. As mentioned in [4] however, guided parsing is algorithm specific, as different parsing algorithms encounter nondeterminism at different steps in their algorithm, and manage nondeterminism in different ways. For this reason, this paper focuses solely on grammar filtering as a more general approach to run-time reduction for context-free parsing algorithms.

There are a number of papers exploring guiding and filtering strategies for parsing algorithms which broadly follow the principles of this paper: use some criteria relating the input s_i to the rules in G in order to exclude some rules during preprocessing or prevent their application during parse time. These "guides" or "oracles' can take the form of any number of complex processes such as preparsing using a different grammar formalism, constructing an extension of the original language using automata, or filtering the grammar using easily precomputed information about the input. See [4], [6] for applications to CFG parsing and [5], [1] for similar principles applied to Tree Adjoining Grammars.

The broadest, most recent catalogue of *filtering* strategies for context-free grammars is presented in [6]. [6] offers a number of filtering protocols anchored by three mutually independent algorithms to find subgrammars, as well as a fourth algorithm they call the 'make-a-reduced-grammar' algorithm - a procedure relying on the removal of rules containing unproductive and unreachable symbols, which can be found in many introductory texts [17] and which merely reduces the size of the CFG, not the expressiveness of the CFL it represents.

The paper's algorithms include **b-filtering**, which relies on eliminating rules containing terminal symbols which are not in s_i, **a-filtering** which uses adjacency criteria to eliminate all rules containing adjacent symbols, which generate terminal symbols that are not adjacent in s_i, and finally **d-filtering**, described further in [4], which relies on the construction of a pair of finite state automata defining regular languages that are supersets of the original CFL in question.

In this paper we introduce run-time improvements to the b-filtering algorithm, ignoring both a-filtering and d-filtering because they rely on adjacency information within the input string and so would not generalize to noise-skipping parsers, an important variant of parsing algorithms for real world applications; additionally the d-filter incurs overhead to produce its regular language supersets of G that [6] concedes may suffer from a combinatory explosion for sufficiently large grammars (though the exact formal properties of this technique are left unexplored by the authors). Furthermore, b-filtering is generally a precursor step to a and d-filtering, so all subsequent filtering strategies are bottle-necked by its run time. An important limitation of b-filtering to note is that it only is able to filter out rules containing a terminal, also called a lexicalized rule. This is especially a problem for any grammars which are already in Chomsky-normal form (CNF) [10] as they tend to have a minimal number of lexicalized rules. CNF is used a preprocessing step for the CYK algorithm, so it is likely that our filtering method, and the ones presented in [6] would have limited effectiveness for speeding up CYK parsers.

We now describe b-filtering as presented in [6], our b-tree filtering algorithm, and empirical and theoretical results showing its improvement over the original.

3 DEFINITIONS

Context-free languages (CFLs) are a formal classification of language within the Chomsky-hierarchy of languages [11]. They are typically defined either as the set of all string sets recognizable by pushdown automata (PDA) (where each automaton defines a language), or they are defined as the set of strings generable by a context free grammar.

A context-free grammar (CFG) is a 4-tuple $G \equiv (V, P, T, S)$, where V is a Vocabulary of string symbols appearing in the grammar, T is a set of terminal strings which can appear as the tokens in strings belonging to the language, P is a list of productions (or rewrite rules - hence the term rewrite system used to describe this style of representation) of the form $A \rightarrow \alpha\beta\gamma...$ where the left-hand side (LHS) is a single nonterminal (formally, an element in V/T) and the right-hand side (RHS) is a sequence (possibly empty) of elements in V, either terminals or nonterminals. Finally, S is a single element in V called the sentential form, or start symbol. Any string accepted by the language must be generable by a series of productions starting at S.

A CFG's size $|G|$ is define as the sum of the lengths of the RHSs for all productions in P. A symbol $\alpha \in V$ is unreachable in G if there is no series of productions starting at S which contains α. A symbol $\beta \in V$ is unproductive if no series of productions $\beta \rightarrow ...$ results in a string containing only terminals from T. A rule is useful if it contains no unproductive or unreachable symbols.

The process of determining whether a given string s is in a CFL is called recognizing, while the process of finding all possible derivations is called parsing. A derivation is described as a parse tree. The set of all parse trees for a string s is called a parse forest.

Finally, parsers are algorithms which take grammars G and input strings s and return parse forests. This paper focuses on the Earley parser, an algorithm whose run time, as with many other general CFL parsing algorithms, is $O(|G| \times |s|^3)$ [6].

See [26] for more details on parse forests, [17] for more details on CFLs and formal languages more broadly, and [28], [21], [30] for more information on CFL parsers.

4 B-FILTERING

b-filtering, when applied to a grammar G and input s works by removing all rules that contain terminals in their RHS that aren't featured in the input s. This method is correct because any rule P with terminal α can only ever successfully be applied to a string containing α. Thus, when parsing s it is safe to remove P if $\alpha \notin s$.

Consider the grammar G below applied to the input $a\ b$:

$$S \rightarrow A\ B \quad (1)$$
$$S \rightarrow C\ B \quad (2)$$
$$A \rightarrow a \quad (3)$$
$$B \rightarrow b \quad (4)$$
$$C \rightarrow c \quad (5)$$

The b-filtering strategy would eliminate the final rule, leaving us with just rules 1-4.

The downside to Bouiller and Sagot's algorithm is that it requires that we check every rule to see whether it contains any terminals not contained in the input. In particular, we will show in section 6 that its best and worst case run time is $\Theta(|P| \times |T|)$, where $|T|$ is the number of terminals in the vocabulary. For very large grammars, this can be an incredibly time consuming and wasteful process - especially if the vast majority of rules contain terminal symbols excluded by the filtering rule. An ideal solution would be one which only touches those productions which we would like to include in our subgrammar - such a solution would approach the lower-bound performance threshold of $O(|P_s| \times |T|)$, where P_{s_s} are the productions in the filtered subgrammar and $|T|$ is the number of terminals in the vocabulary. We call this design principle **counting-production-in** as opposed to the **ruling-productions-out** approach of standard b-filtering.

b-filtering is a particular algorithm that accomplishes what we will refer to as content filtering. The purposes of this paper is not to modify the semantics of content filtering, but instead to provide a faster algorithm to accomplish it. We refer to the method as terminal-tree filtering because it relies on a tree structure to rapidly index into the grammar based on the terminals in the vocabulary.

In lieu of access to the grammars and specific example sentences they use in their work, we will forego direct comparison of the TTF with their benchmarks and instead compare directly to our own implementation of their b-filtering algorithm. This comparison is valid because all of Boullier and Sagot's benchmarks use b-filtering as a preprocessing step and thus their total run time in all experiments is bounded by the b-filtering time. Therefore, since our TTF finds the same subgrammar in a fraction of the run time, it could be used as a superior preprocessing step to the more advanced a and d-filters offered in their work. We will now introduce the terminal-tree filter algorithm.

5 TERMINAL-TREE FILTERING

5.1 Building the Tree

Terminal-tree filtering is motivated by the desire to content-filter the grammar rules by ruling-productions-in, as opposed to b-filtering's comparatively wasteful strategy of ruling-productions-out. The key difference, is that we want to be able to reject subsets of the grammar without having to check every single rule within those subsets rules, allowing us to more closely approach the idealized scenario of a run time proportional to the subgrammar size.

In order to do this, we expend an upfront cost amortized across all queries to construct a binary tree, each node of which contains a set of rules which is a subset of the original grammar G (in experiment 3, we also index this set by the length of the RHS). This tree is then used to determine the proper subgrammar for each input.

Each level of the tree corresponds to a terminal in T. We assume an indexing a_i over the terminals $a_0, ..., a_{n-1} \in T$. We first present naive TTF which chooses an arbitrary indexing and terminates a node when it has a single rule in it. We then optimize this construction with a more complex partitioning of the rules called **early-termination**, where the rules for each node are partitioned into two sets: those which are terminating at the current level, and those which are continuing into the next layer. A rule terminates at the current level if all symbols in its RHS have already been indexed

at this level. If a node has only terminating rules, it is made into a leaf, regardless of how many terminating rules it contains.

The contents of the tree are defined inductively: the root, at level 0, contains all the rules. The children of each node b_i at level i are constructed as follows: if only one rule is contained in b_i, make b_i a leaf, otherwise, partition the rules into two sets: put those which contain a_i into the left child and those which do not contain a_i into the right - obviously all nodes at level n are leaves regardless of the number of rules they contain. The **early-termination** criterion, stipulates that the rules in node b_i are partitioned into terminating rules (those whose deepest aymbol is a_i and non-terminating rules, furthermore b_i is made into a leaf if it contains only a single non-terminating rule or it only contains terminating rules (i.e. if a_i is the deepest symbol contained in its rules).

Again, though it is not essential for the algorithm, in experiment 3, all rules within each node are organized into sets indexed by the length of the RHS.

There are three **structural properties** of note:

(1) Each level consists of some k nodes which contain a combination of mutually exclusive rule sets which are exhaustive the set of all rules $P \in G$.
(2) There are at most $|P|$ leaf nodes in the tree
(3) If $|P| \gg |T|$ then there are $O(|P|)$ total nodes in the tree

5.2 Filtering

Call A_s the set of indices for all a_i that we are filtering out (that is, the vocab symbols not in our input s). The algorithm first calculates j, the deepest symbol contained in s, then recursively traverses the tree, always traversing only on right children for indices l in A_s, until it either reaches a leaf or reaches level j. If we have reached a leaf, we check if its rules contain any symbols in A_s and union all those rules which don't into our working set. If we have gone past level j, then we are assured that all of the rules in the node pass the filter and can be unioned into the working set. For the early terminating tree, in addition to filtering the rules at leaves like mentioned above, we also take the full set of terminating rules at every node we reach if the symbol for that node a_i is not in A_s.

The pseudocode below demonstrates a recursive implementation of the TTF with no early termination, with $\cup$ denoting set union. Where length-filtering is incorporated, this is a set union over the sets whose RHSs are less than or equal to the sentence length, otherwise it's simply a set union of all rules at that node. In the case where early termination is implemented, the algorithm also adds all the rules in the terminating rule set at each node.

Note that for very large grammars, a recursive style implementation of these algorithms will not run in practice. In the git repository for this project, one can find an iterative implementation inspired by continuation-style programming.

The implementation we use can either simultaneously perform length-filtering on the fly or ignore length. This is true for both tree-based filtering and the standard b-filtering we benchmark against. This is because we store the rules in a hash table partitioned by length of the RHS and are able to choose to either only index rules of appropriate length or index rules of any length

The largest rule used in experiment 3 is of length 12, so this technical point is irrelevant for sentences longer than 12 words, so we will not dwell on it too much. Furthermore, TTF query time isn't substantially impacted by whether length-filtering is used.

Algorithm 1 Terminal-tree Filtering

```
1:  A_s ← symbols not in the input
2:  j ← largest index in A_s
3:  set ← ∅
4:  root.Recurse(j, set, 0, A_s)
5:  procedure RECURSE(j, set, level, A_s)
6:      if this.depth > j then
7:          set ← set ∪ this.rules
8:      else if is_leaf(this) ∧ level ∉ A_s then
9:          set ← filter_list(this.rules, A_s)
10:     else if is_leaf(this) ∧ level ∈ A_s then return set
11:     else if level ∉ A_s then
12:         level++
13:         set ← set ∪ this.left.Recurse(j, set, level, A_s)
14:         set ← set ∪ this.right.Recurse(j, set, level, A_s)
15:     else if level ∈ A_s then
16:         level++
17:         set ← node.right.Recurse(j, set, level, A_s)
        return set
18: procedure FILTER_LIST(ruleList, A_s)
19:     set ← ∅
20:     for rule ∈ ruleList do
21:         ruledOut ← false
22:         for terminal ∈ rule.rhs do
23:             if terminal ∈ A_s then
24:                 ruledOut ← true
25:                 break
26:         if ¬ruledOut then
27:             set ← set ∪ rule
        return set
```

Algorithm 2 Terminal-tree Filtering; Early Terminating

```
1:  A_s ← symbols not in the input
2:  j ← largest index in A_s
3:  set ← ∅
4:  root.Recurse(j, set, 0, A_s
5:  procedure RECURSE(j, set, level, A_s)
6:      if this.depth > j then
7:          set ← set ∪ this.terminatingRules
8:          set ← set ∪ this.nonterminatingRules
9:      else if is_leaf(this) ∧ level ∉ A_s then
10:         set ← set ∪ this.terminatingRules
11:         set ← filter_list(this.rules, A_s)
12:     else if is_leaf(this) ∧ level ∈ A_s then return set
13:     else if level ∉ A_s then
14:         set ← set ∪ this.terminatingRules
15:         level++
16:         set ← set ∪ this.left.Recurse(j, set, level, A_s)
17:         set ← set ∪ this.right.Recurse(j, set, level, A_s)
18:     else if level ∈ A_s then
19:         level++
20:         set ← node.right.Recurse(j, set, level, A_s)
        return set
```

6 ANALYSIS

6.1 Terminal-tree Filter is $O(|P| \times |T|)$

We will now provide a construction for a grammar that forces the TTF to consume $O(|P| \times |T|)$ space and achieves a worst-case run time of $O(|P| \times |T|)$ for the TTF. Note that for this grammar, $|G| \approx |T|$. In a separate technical report, we prove that the worst-case run time and memory consumption for the TTF is $O(|P| \times |T|)$, while this section merely proves that these bounds are tight by providing an example grammar which forces such a run time/space consumption[14].

This grammar is constructed on the premise that trees with long, left-branching chains in the tree, create long chains that grow in proportion to the size of the subset of the grammar contained by the leftmost node in the chain, while right-branching chains can create chains of depth proportional to $|T|$ for arbitrarily small subsets of the grammar.

The first claim is true because, if a chain branches left for depth d, and the final node contains k rules, each rule must contain at least d RHS symbols (otherwise they wouldn't be on a left branching subtree). Consequently, the size of the subset of the grammar contained in the bottom most is at least $k \times d$. Thus, even though the chain is of size d, those nodes were not created vacuously, as they must be offset by some node containing an $O(d)$-sized subgrammar.

On the other hand, right branching chains are vacuous. This is so because it's possible for some small number of rules $k > 2$ to sit at the end of a right branching chain of length d as long as those rules contain none of the d symbols corresponding the levels at which the subtree branched right. Of course, some rule must contain those d symbols, otherwise they wouldn't be in the vocabulary at all, therefore for every right-branching chain of length d, there must be some left-branching chain containing a subset of the grammar of size $O(d)$.

The question then becomes, is it possible to create an arbitrary number of right-branching chains which are compensated for by only a fixed, finite number of left-branching chains?

We can accomplish this by starting with a grammar G' whose tree is a complete binary tree of depth $|T'|$. We then augment the original G to create one long, left-branching chain of depth k off the left-most leaf, and then right-branching chains of length k off of every other leaf containing more than one rule.

The initial grammar G' would consist of $|T'|$ symbols, one rule for each size-one subset in the powerset of $|T'|$ and two rules for each subset of size greater than one. For all of the $2^{|T'|} - |T'|$ subsets with more than one element, we produce one rule with them appearing in increasing index order in the RHS and a duplicate rule with them appearing in decreasing order. The resulting grammar has $2 \cdot (2^{|T'|} - |T'| - 1) + |T'|$ productions, and is a full binary tree of depth $|T'|$, containing $2^{|T|} - 1$ total leaf nodes, and $2^{|T'|} - |T'| - 1$ leaf nodes with more than one rule. An abbreviated example with $|T'| = 3$ is below:

$$S \rightarrow a \mid b \mid c \mid a\ b \mid b\ a \mid ... \mid c\ b\ a$$

The initial tree size is $|Tree'| = O(2^{|T'|}) = O(|P'|)$, while the initial grammar size is $|G'| = |T'| + \sum_{i=2}^{|T'|} i \cdot \binom{|T'|}{i} = O(|T'| \times 2^{|T'|}) = O(|P'| \times |T'|)$.

We can then augment G' to produce a new G that adds one arbitrarily long left-branching chain to the leftmost portion of the tree, forcing each of the $2^{|T'|} - |T'| - 1$ leaf nodes with more than one rule to add a long right-branching chain. We do this by adding k new unique symbols to the two rules containing all $|T'|$ of the original symbols. This rule ensures that there are two strings in the language for which filtering requires traversing every node in the tree. We don't add any of the new k symbols to any other rule. The resulting grammar has $|T| = k + |T'| = O(k)$, $|P| = |P'| = O(2^{|T'|}$, $|G| = |G'| + k = O(k)$. On the other hand, the resulting tree now has $|Tree| = |Tree'| + k \cdot (2^{|T'|} - |T'| - 1) = O(k \times |P'|) = O(|T| \times |P|)$ - note that $|P| = |P'|$. The new grammar is abbreviated below:

$$S \rightarrow a \mid b \mid c \mid a\ b \mid b\ a \mid ... \mid c\ b\ a\ \alpha_0\ \alpha_1\ \alpha_2$$

We have already shown that this construction contains $O(|P| \times |T|)$ nodes. It is relatively straightforward to see that for all grammars, filtering an input which contains all of the symbols makes the TTF algorithm visit every node, therefore achieving $O(|T| \times |P|)$ run time. This is due to lines 14 and 15 in Algorithm 1. In the next section we show that the b-filter achieves $O(|G|)$ or $O(|P| \times |T|)$ run time on all grammars, even those for which the resulting tree size is closer to $O(|P|)$, like the binary tree described earlier. In the following section, we will argue why the TTF algorithm has better expected performance than the b-filter. Additionally, in experiment 1, we will empirically compare run times on this worst-case grammar and on the best-case binary tree grammar.

Note that we must perform a call to filter_rules every time we reach a leaf, this only adds an $O(c \cdot |P| \times |T|)$ factor, where c is the largest number of rules in a leaf because there are $|P|$ leaves and it takes $|T|$ steps to filter a rule .This is a worst-case in general, though for this particular grammar, there are only two rules which take $|T|$ to filter (the two at the end of the left-branching chain), so we get a constant factor of $O(|T|)$ added due to filtering, as filtering for all the other leaves will take $O(c \cdot |P|)$ negligible time, which is negligible. This doesn't affect the asymptotic worst-case analysis.

For this grammar, adding the **early-termination** criteria can lower the run time to $O(|P| + |T|) = O(|T|) = O(|G|)$, if we are using an ideal indexing, because none of the right-branching subtrees are present, so worst case only the single length $O(|T|)$ left-branch is traversed in addition to the $O(|P|)$ nodes in the full binary section.

To summarize, worst time run times are below:

$$\begin{aligned}
\text{early TTF} &= |\text{nodes visited}| + |\text{filter rules} \\
&= O(|T|) + O(|T|) \\
&= O(T) \\
&= O(|G|) \\
\text{TTF} &= |\text{nodes visited}| + |\text{filter rules}| \\
&= O(|P| \times |T|) + O(|T|) \\
&= O(|P| \times |T|)
\end{aligned}$$

median run times should be $O(|P|)$ for both variants, because none of the deep chains should be visited. Though it should be noted, that is only in the case of this particular grammar and this particular indexing, in general, the median run time will be the same as the worst run time.

6.2 B-filtering is $\Theta(|G|)$

Unfortunately, [6] provides no implementation or analysis for b-filtering. The most straightforward approach to b-filtering would involve checking every rule to see if it contains any symbols not in the input. This can be done by constructing the set A_s of symbols not in the input and checking for each $\alpha \in A_s$ and for each $p \in P$ whether $\alpha \in p$. This yields a run time proportional to $\Theta(|P| \times |A_s|)$. This run time is a fixed tight-bound for all input strings and input grammars. Thus, although in the worst-case grammar provided above, the b-filter has a theoretical advantage over the TTF, the $O(|P| \times |T|)$ bound for the TTF is loose, and run time can be far below that for many of the strings in the grammar. In later sections, we will show that even on the worst-case-construction grammar, that the TTF achieves a theoretical speed up for all but the two longest strings in that language.

6.3 Typical TTF Runtime is Faster than the Worst-case $O(|P| \times |T|)$

We will argue first that worst-case run times for the TTF are worse than those for the b-filter. We will, however, also argue that empirical performance is far superior for three reasons:

(1) For some worst-case grammars, *typical* sentences in the language show better filter times under TTF than b-filtering.
(2) Good indexing can prevent suboptimal tree shapes like the one given in the previous section
(3) A tree will have $O(|P|)$ nodes whenever $|P| \gg |T|$. Therefore, *typical* grammars will look more similar to full binary trees than to long chains.

We show that **property 1** is true for this grammar, though it is impossible to say what is typical for ALL grammars given that we cannot assign a distribution to the space of all grammars one might see in practice. We discuss how indexing as described in **property 2** can reduce worst-case or average-case run time. Finally, we show that **property 3** always holds and we will argue that it is more typical for $|P| \gg |T|$ than $|T| \gg |P|$ in practical grammars.

A separate technical report[14] for this shows $O(|P| \times |T|)$ is a worst-case upper bound for the run time of the TTF. In an earlier section we showed that this upper bound is actually a tight bound due to the existence of the worst-case grammar we constructed. Despite the worst case bound being inferior to that for b-filtering, we now prove **property 1** for this grammar by showing that $O(|P| \times |T|)$ run-time occurs for precisely two out of the $|P|$ strings in the language. For the other $|P|$ we visit on average some constant fraction of the $O(|P|)$ nodes in the full binary tree portion of the tree. This is so because of our early stopping criterion. Most strings in the language contain only symbols from the initial vocabulary T and therefore never need to search more than T layers deep. Thus on an input that contains all the symbols in T, when we reach layer T, we take all of the rules in all of the right children. This step also takes $O(|P|)$ operations because each leaf contains 1 rule.

Thus our median run time for TTF is $O(|P|) \ll O(|G|)$, where $O(|G|)$ is the run time for b-filters. This is what is meant when we say that *typical* run times are faster for our algorithm. Of course, we can make the list of augmented vocabulary symbols arbitrarily large such that the mean run time for TTF is always greater than the mean run time for b-filtering.

Property 2 is true whenever you can find an index such that long right-branching chains appear at the top of the tree, as this forces there to be a fixed number of such chains. For instance, by indexing all of the k augmented vocabulary symbols before the original elements in T we get one very long left-branching train that terminates in one leaf, and then a long right-branching chain terminating in a full binary tree. Contrast this to our suboptimal indexing from before in which the number of chains was proportional to $|P|$. This would reduce tree size to $O(|T'|)$, giving identical worst-case run time to the b-filter. Of course, in this case, the typical string in the language needs to traverse the entire k-length right-branching chain, increasing the median run-time from $O(|P|)$ to $O(|T|)$. Thus we see that **property 2** allows us to strike balance according to whether we value low median run times or low worst run time. This problem is also made more complicated by the fact that for the documents we are interested in parsing, there is a non uniform distribution over the strings in the language that we'd need to take into account when deciding what behavior we prefer.

Property 3 is true because, for a grammar with a small vocabulary, right-branching chains have essentially constant length. Put differently, each node has at worst a small, constant number of parents with only one child. This means that the tree is approximately a binary tree. The binary tree has $O(|P|)$ leaves and therefore has $O(|P|)$ total nodes.

7 EMPIRICAL RESULTS

Experiments one and two were run in Java 10.0.2 on a System76 Serval with 64 GB Dual-channel DDR4 at 2400 MHz (4× 16 GB) RAM, and 4.2 GHz i7-6700K (4.2 GHz, 8MB Cache, 4 Cores, 8 Threads) processor. Experiment 3 was run on a Dell Precision M2800 with 16 GB RAM and an Intel Core i7 4th Gen 4710MQ (2.50 GHz) processor.

7.1 Experiment 1: Empirical Comparison on a Worst-case Grammar

The first experiment tests a worst-case grammar construction consisting of a core full-binary tree grammar augmented by a large number of vocabulary symbols inserted only into a single production. The vocabulary is intentionally indexed to produce a binary tree with long vacuous right-branching chains at each leaf.

We created grammars according to the formula in the worst-grammar construction in the previous section. We initially chose a small vocabulary from which to create a full-binary tree by generating the powerset of all terminals and producing two rules for each powerset, one with the terminals in increasing lexicographic order and one in decreasing. We then augment the grammar with a large number of symbols which get appended to only the two largest of the original strings. 11 grammars were created with vocabularies of size 11 or 12 and with augmented vocabularies of 20, 40, 60, 80, or 100 thousand. The grammars ranged in size between 62,517 and 249,140. Further details are included in the technical report[14].

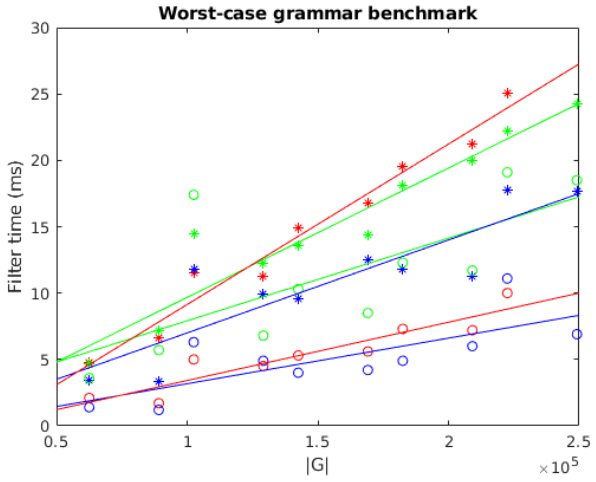

Figure 1: Worst-case Grammar Run Time

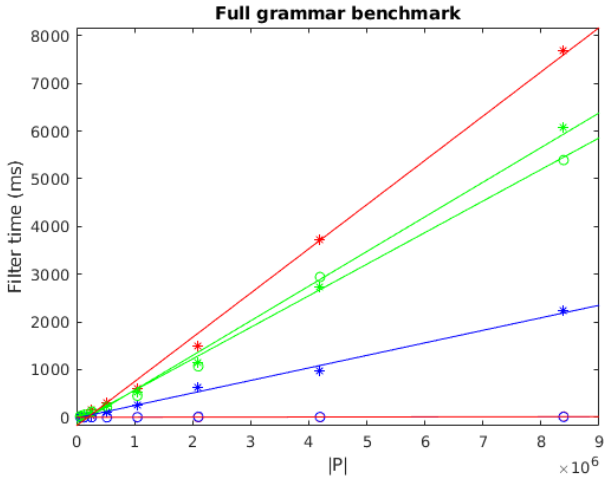

Figure 2: Best-case Grammar Run Time

Displayed in Figure 1 is the filter time for the worst-case grammar described earlier: stars denote run time on the worst-case string in the language (the string containing all symbols), circles denote run time on the median string in the language. Red denotes the naive TTF, green denotes b-filter, and blue is the TTF with early termination.

Empirical results confirm that both variants of the TTF outperform the b-filter on median elements of the sentence and that the early terminating TTF shows better results than the TTF, with a superior constant factor controlling the asymptotic performance. Furthermore, it appears that the early terminating TTF actually has an edge over the b-filter even on the worst case input. Additionally, the b-filter performs similarly under both the worst-case and the median-case. This experiment only had two different sizes of $|P|$ so no meaningful regression was performed against $|P|$ for our run times.

The run times across median/worst length and across all three filters all correlated most strongly with $|T|$ followed by $|G|$, followed by $|P| \times |T|$. The full table of R^2 values is in the technical report[14].

Curiously, the run time curves appear to be best fit by $|G|$ and not by $|P| \times |T|$ (in the case of the worst-length inputs) or $|P|$ (which we should expect for the medians, though we don't have enough different $|P|$'s to run a regression). We would expect $|G|$ to be a better fit than $|P| \times |T|$ for the worst-length times for the early terminating TTF and the b-filter but not for the regular TTF. The latter result can perhaps be explained by the product $|P| \times |T|$ introducing more variance, hurting the empirical correlation. Surprisingly, we find a strong correlation between the median-length run times and $|T|$. It is possible that this results from some artifact related to memory consumption affecting run time, where memory consumption is worse for large $|T|$.

7.2 Experiment 2: Empirical Comparison on a Best-case Grammar

This experiment is similar to the previous except it uses grammars which result in full binary trees, using the method described earlier. Full binary tree grammars were produced with vocabularies $|T| \in [15, 22]$ resulting in sizes between 491,505 and 92 million. A full spec of the grammars is included in the technical supplement[14].

In this experiment, tree size is proportional to $|P|$. $|G|$ is also roughly proportional to $|P|$ because $|T|$ is negligible. This gives us the result that the b-filter and TTF experience asymptotically equivalent run time proportional to $|P|$.

Displayed in Figure 2 is the filter time for inputs on a best-case grammar which produces a full binary tree: as in Figure 1, stars denote run time on the worst string in the language, circles denote run time on the median string in the language. Red denotes the naive TTF, green denotes b-filter, and blue is TTF with early termination.

The empirical results in Figure 2 and Figure 3, which shows a detail of the median-length TTF run times from Figure 2, confirm that all three algorithms, irrespective of input length, have run time proportional to $O(|P|)$. For median length inputs, the TTF algorithms are **400x** faster than the b-filter. We also find that on the worst-length input the b-filter is superior to the naive TTF but is actually outperformed by the early terminating TTF.

7.3 Experiment 3: Empirical Comparison on a Grammar of English

The following experiment was conducted on a proprietary implementation of the TTF in use at Charles River Analytics using a grammar which was a fragment of the English language developed to for efforts related to information extraction in the cybersecurity domain. The implementation described here does not include early terminating TTFs and uses length-filtering to rule out rules which have longer RHSs than the input. The correctness of the length-filtering predicate is only guaranteed when the grammars contain no epsilon productions or if the RHSs contain only terminals.

The test CFG for this experiment was produced by converting a Systemic Functional Grammar [16] of English based on Edouard Hovy's SFG from the Penman project [18].

The resulting grammar has $|P| \approx 13,000,000$ and $|G| \approx 115$ million. Compare this to the two grammars under study by [6]

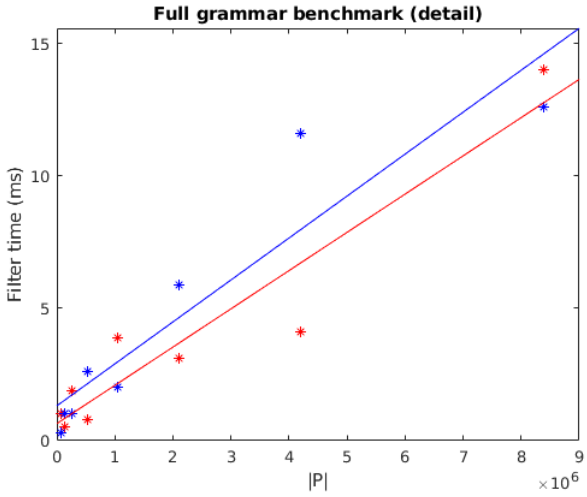

Figure 3: Detail of Median-length Filter Times

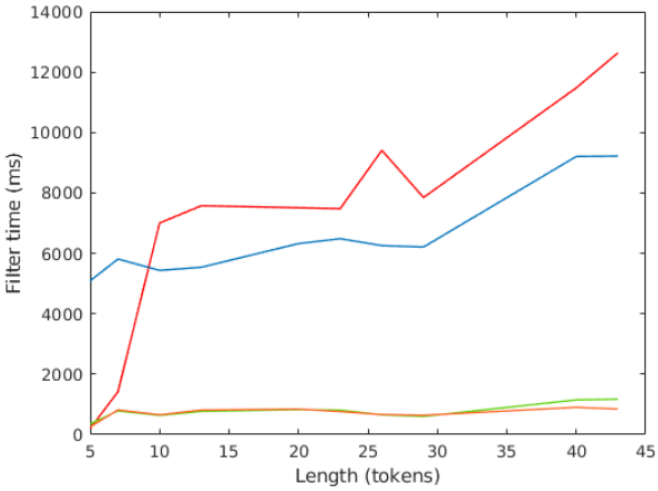

Figure 4: Results on English Language Grammar

which had $\approx$ 500,000 rules each and total size 1 million and 12 million, respectively.

There are only 40 terminals in the vocabulary, making the resulting terminal-tree in our filter bear a structure much more similar to that for the grammar in experiment 2, than the one from experiment 1, in particular it should be closer to a full binary tree and should have few long right-branching chains.

The performance is analyzed on a handpicked set of 10 sentences which can be found in the technical report[14].

Figure 4 contains timed benchmarks comparing filtering time for the TTF coupled with length filtering (orange), the TTF without length filtering (green), b-filtering without length filtering (blue), and b-filtering coupled with length filtering (red).Note the approximate 8-fold speedup of TTFs in the large sentence limit.

Figure 4 demonstrates that b-filtering is competitive only for short sentences and only when combined with length-filtering. Figure 5 displays the ratio of the b-filter's filter time to the TTF's filter time for sentences of various lengths in the English Language Grammar. As you can see, in all length domains studied here, the TTF is at least 7 times faster than b-filtering thus showing a significant improvement over the previous state-of-the-art in [6]. We note that while the b-filtering algorithm exhibits latencies of as long as 8 seconds, our latency remains under a second. This shows that under the prior state-of-the-art, practical parsing of large corpuses would be out of reach for this grammar.

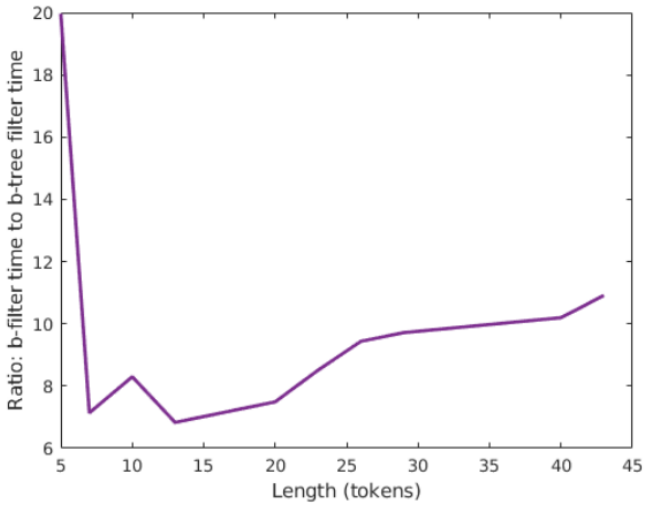

Figure 5: Ratio of B-filter Run Time to TTF Run Time

8 CONCLUSION AND FUTURE WORK

This work has demonstrated a new state-of-the-art filtering technique for massive context free grammar. We have provided a benchmark comparison of a new algorithm, Terminal-tree filtering, which is equivalent to the b-filter used in [6]. The experimental results show that across a range of grammars and workloads designed to mimic the worst-case scenario for our algorithm (experiment 1), the best-case secnario (experiment 2), or a real-life grammar developed at Charles River Analytics for domain-specific NLP tasks (experiment 3), the TTF demonstrates anywhere from slight improvements over the state-of-the-art b-filter presented in [6] on our worst-case grammar to dramatic empirical advantages on grammars exhibiting more plausible properties.

Additionally, we proved a theoretical results explaining the performance of our algorithm. Its performance is optimized on grammars where $|P| \gg |T|$ and in which the queries to the parser tend to be strings randomly drawn from the language, rather than just the longest strings. Both such assumptions are likely to be valid in many use cases. We have also shown that despite our theoretical disadvantages on grammars $|P| \ll |T|$, it is still possible to achieve moderate speed-ups depending on whether the load is primarily strings drawn randomly from the language or solely the longest strings in the language.

Our algorithm tractably filters grammars as large as $|G| \approx 100$ million, several orders of magnitude larger than the largest grammars processed in the most recent comprehensive studies on this subject [6]. Our system dramatically improves the practicality of non-statistical parsing with massive CFGs.

This work expands the upper limits for grammar sizes that can be tractably used in non-statistical parsing applications. Given this increase in the upper limits of available grammar sizes a fruitful area for future work is research on general and robust non-statistical methods for ranking and ordering parses, as well as returning parses in ranked order.

ACKNOWLEDGMENTS

This material is based upon work supported by DARPA under Contract No. FA8750-17-C-0011. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA.

REFERENCES

[1] F. Barthélemy, P. Boullier, P. Deschamp, and É. de la Clergerie. 2001. Guided Parsing of Range Concatenation Languages. In *Proceedings of the 39th Annual Meeting on Association for Computational Linguistics* (Toulouse, France) *(ACL '01)*. Association for Computational Linguistics, USA, 42–49. https://doi.org/10.3115/1073012.1073019

[2] G. Beckers, J. Bolhuis, K. Okanoya, and R. Berwick. 2012. Birdsong Neurolinguistics: Songbird Context-free Grammar Claim is Premature. *Neuroreport* 23 3 (2012), 139–45.

[3] L. Bentivogli, A. Bisazza, M. Cettolo, and M. Federico. 2016. Neural versus Phrase-Based Machine Translation Quality: a Case Study. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Austin, Texas, 257–267. https://doi.org/10.18653/v1/D16-1025

[4] P. Boullier. 2003. Guided Earley Parsing. In *Proceedings of the Eighth International Conference on Parsing Technologies*. Nancy, France, 43–54. https://www.aclweb.org/anthology/W03-3005

[5] P. Boullier. 2003. Supertagging: A Non-Statistical Parsing-Based Approach. In *Proceedings of the Eighth International Conference on Parsing Technologies*. Nancy, France, 55–65. https://www.aclweb.org/anthology/W03-3006

[6] P. Boullier and B. Sagot. 2010. *Are Very Large Context-Free Grammars Tractable?* Springer Netherlands, Dordrecht, 201–222. https://doi.org/10.1007/978-90-481-9352-3_12

[7] D. Chen and C. Manning. 2014. A Fast and Accurate Dependency Parser using Neural Networks. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Doha, Qatar, 740–750. https://doi.org/10.3115/v1/D14-1082

[8] Q. Chen, X. Zhu, Z. Ling, S. Wei, H. Jiang, and D. Inkpen. 2016. Enhanced LSTM for Natural Language Inference. In *ACL*.

[9] N. Chomsky. 1956. Three models for the description of language. *IRE Transactions on Information Theory* 2, 3 (Sep. 1956), 113–124. https://doi.org/10.1109/TIT.1956.1056813

[10] N. Chomsky. 1959. On certain formal properties of grammars. *Information and Control* 2, 2 (1959), 137 – 167. https://doi.org/10.1016/S0019-9958(59)90362-6

[11] N. Chomsky and M.P. Schützenberger. 1963. *The Algebraic Theory of Context-Free Languages*. Computer Programming and Formal Systems, Vol. 35. Elsevier, Amsterdam. 118–161 pages.

[12] S. Clark and J. Curran. 2003. Log-Linear Models for Wide-Coverage CCG Parsing. In *Proceedings of the 2003 Conference on Empirical Methods in Natural Language Processing (EMNLP '03)*. Association for Computational Linguistics, USA, 97–104. https://doi.org/10.3115/1119355.1119368

[13] M. de Marneffe, B. MacCartney, and C. Manning. 2006. Generating Typed Dependency Parses from Phrase Structure Parses. In *Proceedings of the Fifth International Conference on Language Resources and Evaluation (LREC'06)*. European Language Resources Association (ELRA), Genoa, Italy. http://www.lrec-conf.org/proceedings/lrec2006/pdf/440_pdf.pdf

[14] J. Dohmann. 2020. Technical Report. https://github.com/a-fast-filtering-algorithm/A-fast-filtering-algorithm-for-massive-context-free-grammars.

[15] C. Gómez-Rodríguez, I. Alonso-Alonso, and D. Vilares. 2017. How Important is Syntactic Parsing Accuracy? An Empirical Evaluation on Sentiment Analysis. *CoRR* abs/1706.02141 (2017). arXiv:1706.02141 http://arxiv.org/abs/1706.02141

[16] M. Halliday and C. Matthiessen. 2004. *An Introduction to Functional Grammar*. Hodder Education.

[17] J. Hopcroft, J. Ullman, and R. Motwani. 2014. *Introduction to Automata Theory, Languages and Computation*. Pearson education.

[18] E. Hovy. 1990. The Penman Natural Language Project. In *Proceedings of the Workshop on Speech and Natural Language* (Hidden Valley, Pennsylvania) *(HLT '90)*. Association for Computational Linguistics, USA, 430. https://doi.org/10.3115/116580.1138614

[19] M. Johnson. 1998. PCFG Models of Linguistic Tree Representations. *Comput. Linguist.* 24, 4 (Dec. 1998), 613–632.

[20] M. Kay. 2000. Guides and Oracles for Linear-time Parsing. In *Proceedings of the 6th International Workshop on Parsing Technologies (IWPT)*. 6–9.

[21] G. Luger and W. Stubblefield. 2008. *AI Algorithms, Data Structures, and Idioms in Prolog, Lisp, and Java for Artificial Intelligence: Structures and Strategies for Complex Problem Solving* (6th ed.). Addison-Wesley Publishing Company, USA.

[22] P. M. Maurer. 1990. Generating Test Data with Enhanced Context-free Grammars. *IEEE Software* 7, 4 (July 1990), 50–55. https://doi.org/10.1109/52.56422

[23] A. Mishra and S. Jain. 2016. A survey on Question Answering Systems with Classification. *Journal of King Saud University - Computer and Information Sciences* 28, 3 (2016), 345 – 361. https://doi.org/10.1016/j.jksuci.2014.10.007

[24] G. Pullum and G. Gazdar. 1982. Natural Languages and Context-free Languages. *Linguistics and Philosophy* 4, 4 (01 Dec 1982), 471–504. https://doi.org/10.1007/BF00360802

[25] Y. Sakakibara, M. Brown, R. Hughey, I. Mian, K. Sjölander, R. Underwood, and D. Haussler. 1994. Stochastic Context-Free Grammars for tRNA Modeling. *Nucleic Acids Research* 22 (12 1994). https://doi.org/10.1093/nar/22.23.5112

[26] E. Scott. 2008. SPPF-Style Parsing From Earley Recognisers. *Electronic Notes in Theoretical Computer Science* 203, 2 (2008), 53 – 67. https://doi.org/10.1016/j.entcs.2008.03.044 Proceedings of the Seventh Workshop on Language Descriptions, Tools, and Applications (LDTA 2007).

[27] S. Shieber. 1987. *Evidence Against the Context-Freeness of Natural Language*. Springer Netherlands, Dordrecht, 320–334. https://doi.org/10.1007/978-94-009-3401-6_12

[28] S. Shieber, Y. Schabes, and F. Pereira. 1994. Principles and Implementation of Deductive Parsing. *CoRR* abs/cmp-lg/9404008 (1994). arXiv:cmp-lg/9404008 http://arxiv.org/abs/cmp-lg/9404008

[29] T. Strzalkowski, F. Lin, J. Perez-Carballo, and J. Wang. 1997. Building Effective Queries In Natural Language Information Retrieval. In *Fifth Conference on Applied Natural Language Processing*. Association for Computational Linguistics, Washington, DC, USA, 299–306. https://doi.org/10.3115/974557.974601

[30] M. Tomita. 1985. *An Efficient Context-Free Parsing Algorithm for Natural Languages and Its Applications*. Ph.D. Dissertation. USA. AAI8517539.