A fast filtering algorithm for massive context free grammars

Jeremy Dohmann
Kyle Deeds
dohmann@college.harvard.edu
kdeeds@college.harvard.edu
Harvard University

Terry Patten
tpatten@cra.com
Charles River Analytics

ABSTRACT

Context-free languages (CFLs) are a formal class of language with broad applications in natural language processing and related fields [citation needed]. Though much research in the last decade has been focused on strides in statistical [citation needed]and neural [citation needed]methods for natural language parsing, there still exist important use cases for robust non-statistical algorithms [citation needed]. In particular, non-statistical methods are essential for any application in which one may be able to construct a CFG describing the type of structure information they are interested in but lack sufficient hand-annotated data to implement robust statistical parsing algorithms. To solve such problems we turn to classical, non-statistical methods in CFL parsing. In this paper, we focus on using existing parsing algorithms (the Earley algorithm; please see here [citation needed]for an in-depth explanation) to tractably parse inputs using large grammars. Though all the Earley algorithm polynomial in the size of the grammar [citation needed], for sufficiently large grammars parsing is only tractable if the grammar can be filtered somehow. Because there are only a handful of methods to prune grammars without changing the expressiveness of its language [citation needed], it is crucial that filtering strategies be ad-hoc, applied on a per-input basis. Our work is preceded primarily by Boullier and Sagot [?] who provide a suite of ad-hoc filtering methods to practically parse large grammars by determining, in an online fashion, a subset of the original grammar rules applicable for each input. We present a new variant of their basic filtering algorithm which finds the exact same filtered set of grammar rules for a given input, but which achieves a 10-20 fold speedup using a grammar of size $|G| \approx 115,000,000$. Furthermore, we achieve far greater scale, scalably parsing sentences using grammars close to 10 times larger than those they originally studied. Our work is a major contribution because at the grammar size under consideration in this paper, even Boullier and Sagot's suite of algorithms, which are the broadest, most recent contributions to the field, demonstrate unacceptable run times.

KEYWORDS

context-free grammars, parsing, algorithms, filtering

1 DEFINITIONS

Context-free languages (CFLs) [citation needed]are a formal classification of language within the Chomsky-hierarchy of languages [citation needed]. 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.

Regular languages are a class below context-free languages in the hierarchy and have both less expressiveness and more efficient methods for parsing [citation needed]. Regular languages are often defined as either the set of strings sets recognizable by finite state automata (FSA) or by regular expressions.

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 $A \longrightarrow \alpha\beta\gamma\ldots$ 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 CFGs size $|G|$ is define as the sum of the lengths of the RHSs for all productions in P. A symbol $\in V$ is unreachable in G if there is no series of productions starting at S which contains .

A symbol $\in V$ is unproductive if no series of productions $\rightarrow \ldots$ results in a string containing only terminals from T. A rule is useful only if it contains no unproductive and no 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 usually described as a parse tree, and the set of all parse trees for a string s is called a parse forest.

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

See [?] for more details on parse forests, [?] for more details on CFLs and formal languages more broadly, and [?], [?], [?]. [?] for more information on CFL parsers.

2 INTRODUCTION

Context-free languages and application

Context-free languages are a formal class of language originally described by Noam Chomsky [citation needed]and subsequently studied in great depth by linguists and computer scientists due to their usefulness in natural language processing, compilers, and structured information processing more broadly. Parsing natural language is central to machine translation [citation needed], natural language understanding [citation needed], search engines [citation needed], human computer interactions [citation needed]and many more [citation needed].

Though in recent years CFL parsers have ceded some of their supremacy to alternative formalisms such as dependency parsing [citation needed], and combinatory-categorial grammars [citation needed], many of the most prominent syntactic theories in linguistics are based primarily on the study of context-free languages and their intuitive expression, broad generality, and interesting representational and computational properties, permit their application to problems far outside the world of NLP [citation needed].

Among their formal properties, CFLs have been argued to be sufficiently expressive to represent just about every syntactic phenomenon of interest in natural language [citation needed]with limited exceptions [citation needed]. Additionally, CFLs benefit from having a number of subclassifications which impose certain constraints on form and expression in exchange for stronger parsing guarantees [citation needed]. Among these are a host of deterministic context-free languages (those for which there's at most one parse for a given input) such as LL(k) grammars [citation needed]. These grammars are used abundantly in compilers because programming languages have a tendency to be unambiguous anyway and imposing determinism on the language opens up the possibility of linear-time parsing (such as the LL parser) as well as technologies (such as YACC [citation needed]) which enable mass-production of compilers via "parser-generators" [citation needed].

All the expressiveness of general CFLs would be of little practical interest if there weren't tractable parsing algorithms for them. Fortunately, for ambiguous CFLs (the most unrestricted type of CFL), algorithms such as the CKY algorithm [citation needed], the GLR parser [citation needed], and the Earley algorithm [citation needed], which will be the focus of this paper, can parse inputs under an ambiguous CFL in time cubic in the length of the input and in time polynomial in the size of the grammar used to describe it (contrast that to the unambiguous parsers used heavily in compilers which accomplish linear parse times at the cost of reduced expressiveness).

Context-free grammars and size reduction

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:

$$\text{NP} \longrightarrow \text{Noun} \mid \text{Adjective Noun} \mid \text{etc.}$$

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 [citation needed]. These two properties make the task of finding reducing grammar size without reducing expressiveness an open problem. Additionally, grammar reduction is of utmost importance in large-scale parsing applications because algorithms such as the Earley parser quickly become intractably slow and memory intensive as grammar sizes grow [citation needed]. The main reasons for this intractability are that 1.) the run time of the Earley algorithm 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, in general, tends to scale with grammar size [citation needed].

Given the foregoing discussion, 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 seed the grammar (e.g. using probabilistic context-free grammars [citation needed]or neural networks) so as to only pursue a limited subset of potential, partial parses at a time - eliminating less likely alternatives via beam-search methods using probabilities over the likelihoods of certain productions in the rewrite system being realized, conditioned on the context [citation needed]. Though probabilistic grammars have achieved great success within domains where large tagged corpora are available from which to derive meaningful probabilities, much of that thrust in contemporary research has left optimization in non statistically-enhanced parsing under-studied.

There are two main methods by which to improve parse times and reduce the memory overhead associated with pursuing dead-ends in the grammar: guided parsing and grammar filtering. Filtering is a process in which, for each input parsed s_i, an algorithm dynamically 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 elements of the original language. Guiding is a somewhat more sophisticated method, in which, at each step of the parsing phase where nondeterminism can be introduced, instead of exploring all possible rules at once, rule out some which certainly cannot be applied. In many ways, filtering is a degenerate form of guiding, where precisely the same nondeterminism-reduction criterion is applied at every step: don't expand a rule in G unless its also in G_{s_i}. As mentioned in [citation needed]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 primarily 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 during preprocessing or prevent their application during parse time ([?] [?] [?] [?] for CFG parsing, [?] [?] [?] for similar principles applied to Tree Adjoining Grammars).

The broadest, most recent catalogue of filtering strategies for context-free grammars is presented by Boullier and Sagot (2010) [citation needed]. Boullier and Sagot's paper 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 [citation needed]and which merely reduces the size of the CFG, not the expressiveness of the CFL it represents. In contrast to the 'make-a-reduced-grammar' algorithm, which is used simply to cleanup spurious rules and symbols which do not impact the underlying language, the three remaining strategies rely on different ways to scan the input to draw conclusions about what rules could not conceivably be used parse the tokens in the input. The paper's algorithms include the b-filter, 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 [citation needed], which relies on the construction of a pair of finite state automata which describe regular languages which are supersets of the original CFL in question. In this paper we introduce run-timer improvements to the b-filtering algorithm, ignoring both a-filtering and d-filtering because rely on adjacency information within the input string and so would not generalize to noise-skipping parsers, an application addressed elsewhere by the authors [citation needed]; additionally the d-filter incurs overhead to produce its regular language supersets of G that [citation needed]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, [citation needed]treats b-filtering as a precursor step to a and d-filtering, so all subsequent filtering strategies in their work are bottle-necked by its run time.

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

3 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 $a\ b$:

$$S \longrightarrow A\ B \tag{1}$$
$$S \longrightarrow C\ B \tag{2}$$
$$A \longrightarrow a \tag{3}$$
$$B \longrightarrow b \tag{4}$$
$$C \longrightarrow c \tag{5}$$

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, it runs in time $O(|P| \times |s|)$. 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 $O(|P_s| \times |s|)$, where P_s is the set of productions in the filtered subgrammar. 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 logic of content filtering, but instead to provide a faster algorithm to perform it.

As mentioned in the previous section, Boullier and Sagot's paper introduces two, more computationally expensive filtering techniques called a-filtering and d-filtering. Due to the expense of computing a-filtering and d-filtering, all of their experimental results rely on first running a round of b-filtering followed by some policy consisting of interleaved rounds of a and d-filtering. The present work concerns itself with a new algorithm for accomplishing the exact same filtering result as b-filtering, but in time proportional to $O(|P_s| \times |s|)$. We refer to the method as b-tree-filtering as it relies on a tree data structure to rapidly index into the grammar.

In lieu of access to the grammars and specific example sentences they use in their work, we will forego direct comparison of b-tree-filtering with their benchmarks and instead compare directly to our own implementation of b-filtering. 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 b-tree-filter 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.

Additionally, Boullier and Sagot measure the total precisions of the resulting filtered grammar under all of their filtering policies, where precision is defined to be the number of rules used to parse an input divided by the size of the subgrammar found. Because b-tree-filtering and b-filtering find the same subgrammar, doing a side-by-side comparison would not be especially meaningful. For the sake of parsimony, however, we include precision in our discussion below.

We now briefly introduce the grammar used in this experiment, discuss some of its properties, then present b-tree-filtering and its performance over the grammar.

4 OUR GRAMMAR

Our test CFG was produced by converting a Systemic Functional Grammar of English based on HallidayâĂŹs grammar from the Penman project [?]

The resulting grammar has around 13 million rules. The total grammar size $|G|$, measured by the sums of the lengths of the right hand sides is $115,738,333$. Compare this to the two grammars under study by [?] which had $\approx 500,000$ rules each and total size 1 million and 12 million, respectively. The structure of the Halliday grammar is relatively flat and much of the rules arise from different ways of permuting otherwise equivalent RHSs. In particular, there are $8,975,867$ unique RHSs and $35,228$ unique sets of RHS elements. A consequence of this structure is that it is particularly well suited to both length-based filtering when input sentences are short (a basis for filtering not mentioned in [?] because their study permits grammars with ϵ-productions, while we ours does not) and b-filtering. Note that length-based filtering could not be performed on a grammar containing empty productions as in any grammar where nonterminals don't necessarily yield at least one output symbol, there is no relationship between the number of nonterminals on the RHS of a rule and its potential applicability to an input or some fixed size. The grammar (and its accompanying vocabulary) we have constructed is incomplete and canâĂŹt handle all possible English constructions. For that reason we will be analyzing performance on the following sentences only:

- the distributor registered this domain
- this domain was registered by the distributor
- this domain in the account was registered by the distributor
- this domain in the account was registered by the distributor in the domain
- the use of the nicknames in the decryption key is evidence that the account was the distributor of these samples
- the use of the nicknames of this domain in the decryption key is evidence that the account was the distributor of these samples
- the use of the nicknames of this domain in the decryption key in the record is evidence that the account was the distributor of these samples
- the use of the nicknames of this domain in the decryption key in the record of the activity is evidence that the account was the distributor of the software
- the use of the nicknames of this domain in the decryption key in the record of the activity is evidence that the account was the distributor of the software and that it was registered by the communication of the distributors
- the use of the nicknames of this domain in the decryption key in the record of the activity is evidence that the account was the distributor of the software and that it was registered by the communication of the distributors in these discussions

In order to get a sense of how many rules of this grammar are truly applicable for each input, we filter first by length, then by content, and finally, by removing nonterminals which were made unproductive or unreachable by the previous two steps of filtering. Below is a graph relating length of the input to the number of rules left after all stages of filtering.

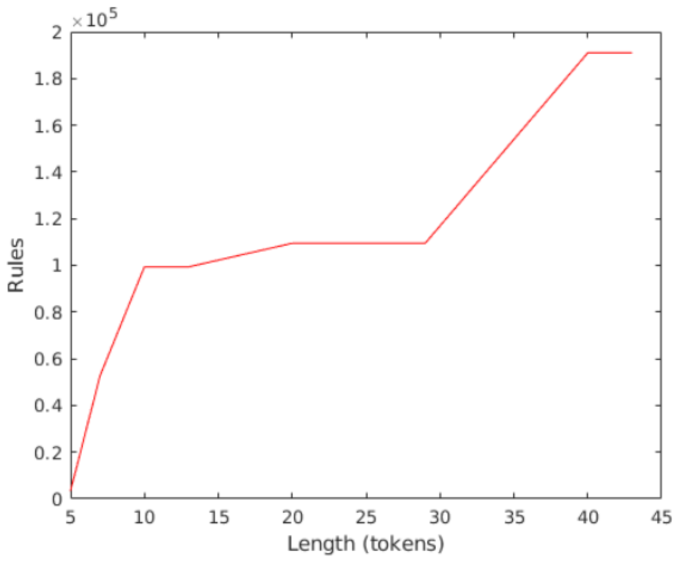

Figure 1: Number of rules resulting after content-filtering. Content-based filtering as exemplified in Boullier and Sagot's b-filtering and our b-tree filtering accomplishes close to 100 fold reduction of grammar size on sentences 45 words long

As you can see, our grammars respond incredibly well to content filtering, achieving a factor of 100 reduction in grammar size on even the longest test sentence.

We will now introduce b-tree-filtering, and show an empirical comparison to b-filtering, and a combination of length-filtering and b-filtering.

5 B-TREE-FILTERING

b-tree-filtering is motivated by the desire to content-filter the grammar rules by ruling-productions-in, as opposed to the b-filtering's comparatively wasteful strategy of ruling-productions-out. The key difference, is that we want the number of operations performed to be proportional to the number of rules in the resulting subgrammar *not* to the size of the original grammar.

In order to do this, we construct a binary tree, each node of which contains a set of rules indexable by the length of the RHS. Note again here, that the incorporation of length-filtering prevents direct comparison to the exact variant b-filtering in Boullier and Sagot's work, though for clarity we present a version of b-filtering that also relies on data structures indexable by length.

Each level of the tree corresponds to a terminal in T. Thus, we assume an indexing a_i over the terminals $a_0, \dots, a_{n-1} \in T$. The indexing is arbitrary, though in practice it is a good idea to randomize the order of terminals.

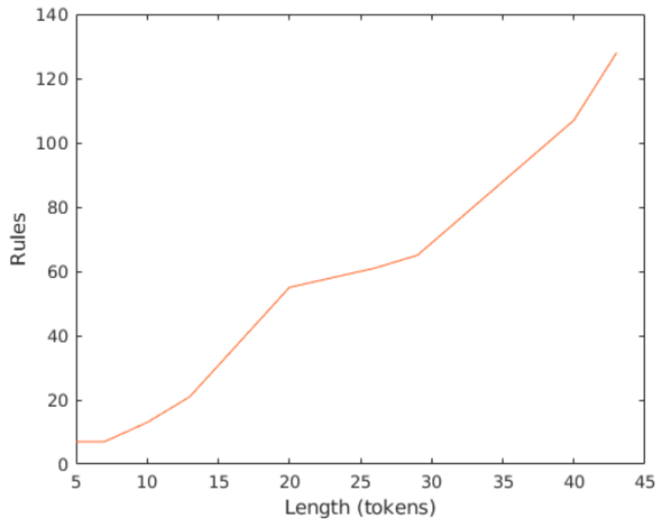

Figure 2: Number of rules used in successful parse (gold-standard). Though our content-filtering methods reduce the grammar to around 100,000 candidate rules, the actual parses for our sentences use on the order of 100 rules for long sentences, giving us relatively small precisions.

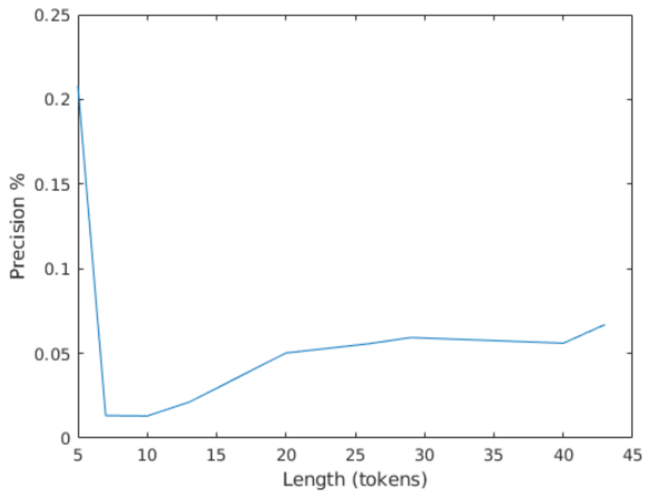

Figure 3: Precision expressed as % as measured by number of rules in the gold-standard divided by number of rules in the subgrammar.

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, give it no children, otherwise, partition the rules into two sets: those which contain a_i and those which do not contain a_i. Note that we could've used any stopping-criterion to define the leaf nodes, but cutting off at 1 rule is intuitive and easy to analyze.

The consequence of this construction is that for all nodes at level i that have children the left child (potentially null) consists of all those rules which do not have a_i, and the right child consists of all those rules which do have a_i. Again, though it is not essential for the algorithm, in our implementation, 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 are mutually exclusive and exhaustive of the set of all rules in the grammar.
(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

Call F the set of indices for all a_i that we are filtering out. Note that because each level contains all the rules, we never need to concern ourselves with nodes below level j where j is the maximum index in S

Consequently, the algorithm first calculates j, then recursively traverses the tree, always traversing only on right children for indices l in S, until it either reaches a leaf or passes level j. Once an execution path reaches one of the above two terminating conditions, it unions the rules in the right child of the present node (because these are guaranteed to not contain any of the terminals from S) to the working set and terminates that branch of the search. Any time we reach a leaf node we just directly check if its singleton rule contains any missing symbols in it.

The pseudocode below demonstrates the recursive search, with denoting set union. In the case where length-filtering is included then 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.

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 Hashtable 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 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, b-tree filter time isnâĂŹt substantially impacted by whether length-filtering is used.

Algorithm 1 b-tree-filtering

```
1: $S \leftarrow$ symbols not in the input
2: $max \leftarrow$ largest index in $S$
3: $set \leftarrow \emptyset$
4: $root.Filter(max, set, 0, S)$
5: procedure FILTER(max, set, depth, S)
6:     if $this.depth \geq depth$ then
7:         $set \leftarrow set.rules$
8:     if $this.rules.length = 1 \wedge depth \notin S$ then
9:         $set \leftarrow this.rules$
10:    if $depth \notin S$ then
11:        $set \leftarrow this.left.Filter(max, set, depth + 1, S)$
12:        $set \leftarrow this.right.Filter(max, set, depth + 1, S)$
13:    if $depth \in S$ then
14:        $set \leftarrow node.right.Filter(max, set, depth + 1, S)$
       return $set$
```

Analysis

oofda

Results

Figure 4 contains timed benchmarks comparing filtering time for b-tree filtering coupled with length filtering, b-tree filtering without length filtering, b-filtering without length filtering, and b-filtering coupled with length filtering.

Note that b-tree-filtering and b-filtering both have relatively stable run times as a function of sentence length for the sentences tested here. At short sentence lengths b-filtering is competitive with b-tree-filtering only when combined with length-filtering. In figure 5 we see that in all length domains studied here b-tree filtering is at least 7 times faster than b-filtering thus showing an extreme improvement over the simpler approach to content-based filtering presented in [1]. A very important point is that our filtering speed remains under a second, even for incredibly long sentences, while Boullier and Sagot basic filtering algorithm exhibits unacceptably large latencies of as long as 8 seconds.

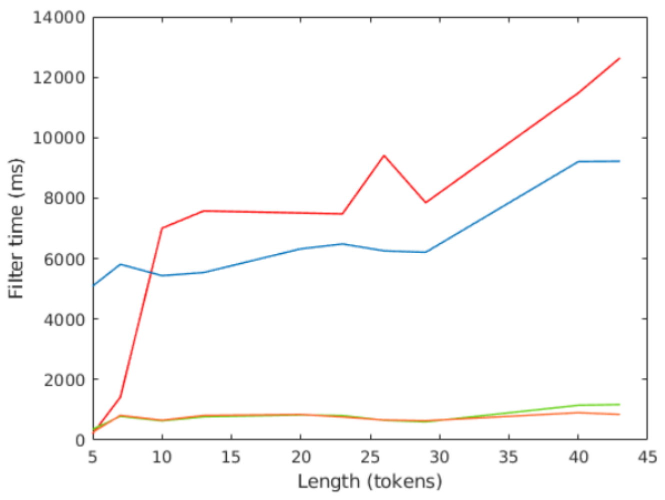

Figure 4: Filtering time in (ms) for sentences of varying lengths using the four methods detailed here. Orange: b-tree filtering with length filtering, Green: b-tree filtering without length filtering, Red: b-filtering with length filtering, Blue: b-filtering without length filterings. Note the approximate 8-fold speedup of b-tree filters in the large sentence limit.

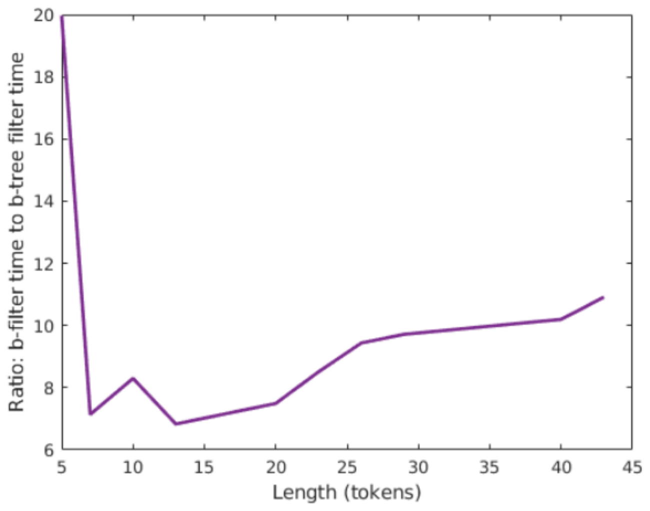

Figure 5: At least 7-fold speedup when using b-tree filtering with no length filter compared with [1]âĂŹs b-filtering. Our method does increasingly well for longer sentences.

The asymptotic relationship between run time of the b-tree filter and the number of rules in the filtered subgrammar appears roughly linear for our example grammar, supporting the argument that, in expectation, the run time is linear in the size of the subgrammar, though we lack a sufficiently large number of samples from the language to rigorously test this hypothesis.

6 CONCLUSION

This work has demonstrated an extension to the most recent literature of filtering techniques for massive context free grammars. We have provided a benchmark comparison of a new algorithm, b-tree filtering, which is equivalent to the preprocessing filter used in [1]. Figure 7 show our algorithm runs anywhere between 7 and 20 times faster than Boullier and SagotâĂŹs basic filter on a grammar roughly 10 times the size of those investigated in their research. We have shown that our algorithm provides reasonable filtering times even for very long input sentences.

ACKNOWLEDGMENTS

Yo

REFERENCES