JMLR: Workshop and Conference Proceedings 34:139–152, 2014 Proceedings of the 12th ICGI

Grammatical Inference of some Probabilistic Context-Free Grammars from Positive Data using Minimum Satisfiability

James Scicluna JAMES.SCICLUNA@UNIV-NANTES.FR
Colin de la Higuera CDLH@UNIV-NANTES.FR
Université de Nantes, CNRS, LINA, UMR6241, F-44000, France

Editors: Alexander Clark, Makoto Kanazawa and Ryo Yoshinaka

Abstract

Recently, different theoretical learning results have been found for a variety of context-free grammar subclasses through the use of distributional learning (Clark, 2010b). However, these results are still not extended to probabilistic grammars. In this work, we give a practical algorithm, with some proven properties, that learns a subclass of probabilistic grammars from positive data. A minimum satisfiability solver is used to direct the search towards small grammars. Experiments on typical context-free languages and artificial natural language grammars give positive results.

Keywords: Grammatical Inference, Probabilistic Context-Free Grammars, Minimum Satisfiability, Congruential Grammars, Small Grammars

1. Introduction

Recently, a series of positive theoretical learning results were obtained for a variety of context-free grammar subclasses under different learning models (Clark and Eyraud, 2007; Yoshinaka, 2008; Clark, 2006, 2010a; Shibata and Yoshinaka, 2013; Clark, 2013). A common theme of these CFG subclasses is that their non-terminals are directly linked to features of the underlying language (Clark, 2010b). In fact, it is precisely this relationship that makes such classes 'learnable' using distributional learning techniques.

To our knowledge, no work has been done to extend these learnability results to learning probabilistic grammars. The problem is that learning PCFGs requires that the learned structure (i.e. the underlying CFG) is somehow close to the target structure. With the only exception of (Clark, 2013), all the results cited above (and similar other results) give weak learning algorithms, which means that no importance is given to the learned structure because any grammar found that generates the target language is good enough. In reality, practically all of these weak learning methods end up returning very large grammars containing all combinations of valid production rules. Assigning probabilities to such grammars is not the ideal way of learning probabilistic grammars.

In this paper, we give a practical algorithm for learning a subclass of probabilistic grammars. The non-stochastic version of this subclass is very similar to the one defined in (Clark, 2010a). Although we do not give a theoretical learnability result, we still prove some properties related to the algorithm. The main feature of our algorithm is that, unlike weak learners, it tries to find small grammars. We evaluate our algorithm on typical context-free

languages and artificial natural language grammars. The results we obtain are competitive with the state-of-the-art.

2. Preliminaries

2.1. Grammars and Languages

A *context-free grammar* (CFG) is a 4-tuple $\langle N, \Sigma, P, I \rangle$, where N is the set of non-terminals, Σ the set of terminals, P the set of production rules and I a set of starting non-terminals (i.e. multiple starting non-terminals are possible). The language generated from a particular non-terminal A is $L(A) = \{w | A \stackrel{*}{\Rightarrow} w\}$ and the language generated by a grammar G is $L(G) = \bigcup_{S \in I} L(S)$. A CFG is in *Chomsky Normal Form (CNF)* if every production rule is of the form $N \to NN$ or $N \to \Sigma$.

A *probabilistic context-free grammar* (PCFG) is a CFG with a probability value assigned to every rule and every starting non-terminal. The probability of a leftmost derivation from a PCFG is the product of the starting non-terminal probability and the production probabilities used in the derivation. The probability of a string generated by a PCFG is the sum of all its leftmost derivations' probabilities. The stochastic language generated from a PCFG G is $(L(G), \phi_G)$, where ϕ_G is the distribution over Σ^* defined by the probabilities assigned to the strings by G. For a PCFG to be *consistent*, the probabilities of the strings in its stochastic language must add up to 1 (Wetherell, 1980). Any PCFG mentioned from now onwards is assumed to be consistent.

2.2. Congruence Relations

A *congruence relation* $\sim$ on Σ^* is any equivalence relation on Σ^* that respects the following condition: if $u \sim v$ and $x \sim y$ then $ux \sim vy$. The congruence classes of a congruence relation are simply its equivalence classes. The congruence class of $w \in \Sigma^*$ w.r.t. a congruence relation $\sim$ is denoted by $[w]_\sim$. As a consequence of the condition imposed on congruence relations, it is always true that $[uv] \supseteq [u][v]$ for any $u, v \in \Sigma^*$.

Two strings u and v in the same congruence class are *relatives* if either $u = v$ or there are $k \geq 2$ congruence classes $[w_1]$, $[w_2]$, ..., $[w_k]$ s.t. u and v are both in $[w_1][w_2] \dots [w_k]$ (i.e. u and v are contained in the concatenation of these congruence classes). By $[uv] \supseteq [u][v]$, we can restrict the definition w.l.o.g. to only $k = 2$. Note that the relatives relation is reflexive, symmetric and it does not need to be transitive. This is known as a *tolerance relation* (Bartol et al., 2004). If we understand a tolerance as an undirected graph (in our case, nodes are the strings in the same congruence class and edges are between strings that are relatives), its *blocks* correspond to the graph's maximal cliques (Bartol et al., 2004).

The set of *contexts* of a substring w with respect to a language L, denoted $Con(w, L)$, is $\{(l, r) \in \Sigma^* \times \Sigma^* \,|\, lwr \in L\}$. Two strings u and v are *syntactically congruent* with respect to L, written $u \equiv_L v$, if $Con(u, L) = Con(v, L)$. This is a congruence relation on Σ^*. The *context distribution* of a substring w w.r.t. a stochastic language (L, ϕ), denoted $C_w^{(L,\phi)}$, is a distribution whose support is all the possible contexts over alphabet Σ (i.e. $\Sigma^* \times \Sigma^*$) and is defined as follows:

$$C_w^{(L,\phi)}(l, r) = \frac{\phi(lwr)}{\sum_{l', r' \in \Sigma^*} \phi(l'wr')}$$

Two strings u and v are *stochastically congruent* with respect to (L, ϕ), written $u \cong_{(L,\phi)} v$, if $C_u^{(L,\phi)}$ is equal to $C_v^{(L,\phi)}$. This is a congruence relation on Σ^*.

2.3. Congruential Grammars

Clark (2010a) defines *Congruential CFGs* (C-CFGs) as being all the CFGs G which, for any non-terminal A, if $u \in L(A)$ then $L(A) \subseteq [u]_{\equiv_{L(G)}}$ (where $[u]_{\equiv_{L(G)}}$ is the syntactic congruence class of u w.r.t. the language of G). This class of grammars was defined with learnability in mind. Since these grammars have a direct relationship between congruence classes and the non-terminals, their learnability is reduced to that of finding the correct congruence classes (Clark, 2010a).

This class of grammars is closely related to the class of NTS-grammars (Boasson and Sénizergues, 1985). NTS grammars are all C-CFGs, but not vice versa. However, it is not known whether languages generated by C-CFGs are all NTS-languages (Clark, 2010a). It is known that the class of languages describable using C-CFGs is a subclass of deterministic context-free languages and contains the regular languages, the substitutable (Clark and Eyraud, 2007) and k-l-substitutable context-free languages (Yoshinaka, 2008), the very simple languages and other typical CFLs such as the Dyck language (Boasson and Sénizergues, 1985).

We define a slightly more restrictive class of grammars, which we shall call *Strongly Congruential CFGs* (SC-CFGs). A CFG G is a SC-CFG if, for any non-terminal A, if $u \in L(A)$ then $L(A) = [u]_{\equiv_{L(G)}}$. The probabilistic equivalent of this is the class of *Strongly Congruential PCFGs* (SC-PCFGs), defined as all the PCFGs G which, for any non-terminal A, if $u \in L(A)$ then $L(A) = [u]_{\cong_{(L(G),\phi)}}$. In other words, the strings derived from non-terminals of these grammars are exactly *equal* to the language's congruence classes (i.e. not just *contained* in the congruence classes, as is the case for C-CFGs).

The languages UC1, UC2, UC5, UC7, AC1, AC2, AC3, AC4 in Section 4 can all be described using a SC-CFGs. However, languages UC3, UC4 and UC6 cannot. In fact, the languages UC3, UC4 and UC6 are all made up of an infinite union of congruence classes. This is why they cannot be described using a SC-CFG or even a C-CFG.

2.4. Smallest Grammar

There are different ways of defining the size of a grammar. Five possible ways are: `N`: Number of non-terminals; `P`: Number of production rules; `N+P`: Number of non-terminals + production rules; `RHS`: Total number of symbols in the RHS of every rule; `RHS+P`: Total number of symbols in the RHS of every rule + production rules

We denote the size of a grammar G in general as $|G|$. When necessary, we will specify how the size of the grammar is calculated with reference to the 5 options above.

A *smallest grammar* for a language L in a class of grammars $\mathcal{G}$ is any grammar $G \in \mathcal{G}$ such that $L(G) = L$ and there exists no $G' \in \mathcal{G}$ with $L(G') = L$ and $|G'| < |G|$. There can be multiple smallest (S)C-CFGs in CNF for the same language.

3. Learning SC-PCFGs in CNF from Positive Data

In this section, we describe a practical algorithm for learning SC-PCFGs in CNF from positive data. Although we do not give a theoretical learnability result, we still prove some properties related to this algorithm.

The first step (described in section 3.1) classifies all the substrings of the given sample into classes according to the stochastic congruence relation. The decisions taken on whether to put two substrings in the same class or not are based on the distance between the empirical context distributions of these strings. The algorithm tries to minimize as much as possible the number of times such decisions need to be taken. The second step of the algorithm (described in section 3.2) builds a *small* grammar from the congruence classes learned. Assuming that the first step returns an exact solution, we show that the learned grammar will be at most twice the size of the smallest SC-PCFG in CNF that generates the target language. With the same assumption, we also show that the learned grammar generates a language which is equal or contained in the target language. Probabilities are assigned to the learned grammar using the standard EM algorithm for PCFGs i.e. the Inside-Outside algorithm (Lari and Young, 1990). In section 3.3 we give a procedure that generalizes the learned (non-stochastic) grammar in case that we find a string (not in the sample) that is in the target language and not accepted by the learned grammar.

Throughout this section, G is taken to be the target SC-PCFG in CNF generating a stochastic language (L, ϕ). S is the finite sample given as input. We assume that S has been i.i.d. generated from (L, ϕ) s.t. S is structurally complete w.r.t. G. By structural completeness we mean that every production rule of G is used at least once in generating the sample S.

3.1. Learning the Congruence Classes

Algorithm 1 describes how the congruence classes are induced. The algorithm loops through all the pairs of substrings and decides for each pair whether the substrings should be merged or not. These pairs are ordered according to $\prec$, which can be any ordering on Σ^* such that for any two strings u and v, if u is a substring of v then $u \prec v$. Note that the algorithm does not backtrack from its decisions (i.e. whenever two strings are merged, they are never put back in separate classes).

Substrings are merged in the procedure `CongruenceClosure`. This procedure also finds and merges other pairs of substrings that should logically be merged. These are found by repeatedly applying the rule which congruence classes should follow: if u and v are in the same class and x and y are in the same class then uv and xy should be in the same class.

ECD is the *empirical context distribution*. When defined over a string $w \in Subs(S)$, it is simply a probability distribution over all the contexts of w in S, where the probability for a context (l, r) is the number of occurrences of lwr in the sample divided by the number of occurrences of w. This definition is extended to congruence classes by treating each substring in the class as one substring (i.e. the sum of occurrences of lw_ir, for all w_i in the congruence class, divided by the sum of occurrences of all w_i). The closer the distance between the empirical context distributions of classes $[u]$ and $[v]$ is, the more confident we can be that $[u]$ and $[v]$ should be merged into one congruence class. Any distance between probability distributions can be used here. In our experiments, we used the L1-distance.

Algorithm 1: Learning the congruence classes

Input: Sample S, distance function between distributions `dist`, distance threshold δ, frequency threshold n

Output: The congruence classes $\mathcal{CC}$ over the substrings of S

1 $Subs \leftarrow$ Set of all substrings of S ;
2 $\mathcal{CC} \leftarrow \{\{w\} \mid w \in Subs\}$;
3 $Pairs \leftarrow \{(u, v) \mid u, v \in Subs\ ,\ u \neq v\ ,\ u \prec v\}$;
4 Order $Pairs$ according to v (w.r.t. $\prec$) and if equal according to u (w.r.t. $\prec$) ;
5 **foreach** $(u, v) \in Pairs$ **do**
6 | **if** $(u \neq \texttt{min}_{\prec}[u])$ **or** $(v \neq \texttt{min}_{\prec}[v])$ **then continue** ;
7 | **if** $(\sum_{x \in [u]} |S|_x < n)$ **or** $(\sum_{y \in [v]} |S|_y < n)$ **then continue** ;
8 | **if** $\texttt{dist}(ECD([u]), ECD([v])) < \delta$ **then**
9 | | $\mathcal{CC} \leftarrow \texttt{CongruenceClosure}(u, v, \mathcal{CC})$;
10 | **end**
11 **end**
12 **return** $\mathcal{CC}$;

In line 6 and line 7, the algorithm leaves the two substrings in their current congruence class. In line 6, this is done because at least one of the substrings is not the smallest string (according to $\prec$) in its own congruence class. In this case, the outcome of a previous decision (precisely the decision taken on the pair of smallest substrings in both classes) determined the state of the current substrings through `CongruenceClosure`. In line 7, the algorithm avoids merging whenever one of the congruence classes is too sparse for a good approximation for ECD to be calculated. Sparse congruence classes should be merged though the logically deduced merges taken by `CongruenceClosure`. Note that the notation $|S|_x$ is the number of occurrences of substring x in the sample S. Also note that $[u]$ and $[v]$ in the algorithm are congruence classes in $\mathcal{CC}$ (i.e. the finite number of congruence classes over all substrings of the sample, and not over Σ^*) and $\texttt{min}_{\prec}[u]$ and $\texttt{min}_{\prec}[v]$ are simply the smallest strings w.r.t. $\prec$ in $[u]$ and $[v]$ respectively.

Proposition 1 *If all decisions taken by the algorithm are correct, then the merge decisions taken to identify a target congruence class $[w]$ are exactly the merges between the smallest substrings (w.r.t. $\prec$) in the relative blocks of $[w]$ (i.e. at most, the number of relative blocks minus 1)*

Proof: If two strings u and v in the same target congruence class are relatives, then we can split u and v, $u = u_1u_2$ and $v = v_1v_2$, s.t. u_1 is congruent to v_1 and u_2 is congruent to v_2. Clearly, u_1 and u_2 (resp. v_1 and v_2) are smaller than u (resp. v) w.r.t. $\prec$. Therefore, the decisions to merge u_1 with v_1 and u_2 with v_2 *precede* the decision to merge u with v. If these preceding decisions are taken correctly, then u will be merged with v as a logical consequence (i.e. u will be merged with v in `Congruence_Closure`). This is because if $u = u_1u_2$, $v = v_1v_2$, $u_1 \sim v_1$ and $u_2 \sim v_2$ then $u \sim v$. Therefore, we do not need to take merge decisions on strings that are relatives. The only merge decisions we need to take are between non-relative strings. Merging two strings from different relative blocks will

consequently result in all the strings in the two blocks to be merged together. Therefore, the only merge decisions needed are between the smallest strings in the relative blocks.

3.2. Building the Grammar

Using the congruence classes learned from the previous step, we can build a *primitive grammar* G' as follows (exactly as Clark (2010a) does):

$N = \mathcal{CC}$
$\Sigma = \Sigma(S)$ (i.e. the alphabet of S)
$P = \{[w] \to [u][v] \mid w \in Subs(S);\ w = uv;\ |u|, |v| > 0\} \cup \{[a] \to a \mid a \in \Sigma\}$
$S = \{[w] \mid w \in S\}$

Proposition 2 *Assuming that the congruence classes obtained from the previous step are correct, then the primitive grammar G' will generate the target language*

Proof: Since it is true in general that $[uv] \supseteq [u][v]$ and $a \in [a]$ then G' cannot generate strings which are not in the target language. On the other hand, the structural completeness assumption ensures that a subset of the production rules of G' are isomorphic to the target grammar, which means that G' generates at least the target language.

If we simply want to learn any CFG that generates the target non-stochastic language, then G' is good. However, in our case we are interested in learning the stochastic language. This means that we have to find a CFG for which a probability assignment using the standard EM algorithm for PCFGs (Lari and Young, 1990) yields a stochastic language close to the target one. Using an Occam's razor argument, a good candidate grammar would be the smallest SC-CFG in CNF generating $L(G')$. However, finding such a grammar is an NP-Complete problem. To prove this, we show that this problem is at least as difficult as the Smallest CNF Grammar Problem, which is a known NP-complete problem (Rytter, 2003; Charikar et al., 2005)

Proposition 3 *The **Smallest CNF Grammar Problem**, which is the problem of finding a smallest CFG in CNF (where the size of the grammar can be either `RHS` or `RHS+P`) that generates exactly one given string w is NP-Complete (Rytter, 2003; Charikar et al., 2005)*

Theorem 4 *Given a SC-CFGs in CNF generating L, the problem of finding a smallest grammar for L in the class of SC-CFGs in CNF is NP-Complete.*

Proof: First of all, we can easily construct a SC-CFG in CNF generating one string w. Secondly, and most importantly, any smallest CFG in CNF generating only one string must be a SC-CFG since every non-terminal must generate exactly one congruence class. This is because each non-terminal must generate only one substring of w (otherwise the grammar would not generate only w) and no two non-terminals can generate the same substring (otherwise they would be redundant and thus the grammar will not be a smallest one). Thirdly, the size of a smallest CNF grammar in general in terms of `RHS` is related to the size in terms of `P` by the formula: *`RHS`* = *2`P`* - $|\Sigma|$ and also `P` will be exactly equal to `N` (because every non-terminal is on the LHS of a rule exactly once). Therefore, a grammar is the smallest irrespective of the method used to calculate its size (`N`, `P`, `P+N`, `RHS` or `RHS+P`).

Therefore, from all this we can conclude that the Smallest CNF Grammar Problem is polynomially reducible to the problem described in this theorem.

An alternative is to find a small grammar that is bounded in size w.r.t. the smallest possible grammar. This is in fact the approach we take.

So, our task is to decide which non-terminals/rules are to be chosen from the primitive grammar such that they form a small grammar equivalent to the target. We start by assigning a Boolean variable to every non-terminal/rule which will be true if we decide to choose that non-terminal/rule and false otherwise. We denote non-terminal variables as $N_{[x]}$, $[x]$ being the congruence class as a non-terminal, and rule variables as $R_{[y],[z]}$, representing a CNF rule whose RHS is $[y][z]$ (note that a SC-CFG cannot have two different rules with the same RHS).
We can define the following constraints on these variables:

- Any non-terminal variable whose congruence class contains strings known to be in the target language has to be true (these will be the starting non-terminals).
- Any non-terminal variable whose congruence class contains symbols from Σ has also be true because we need them to define rules of the form $A \to a$.
- If a rule variable $R_{[u],[v]}$ is true, then it is clear that $N_{[u]}$ and $N_{[v]}$ have to be true.
- If $N_{[w]}$ is true, then for every string $w \in [w]$, there must be at least one split of w into u and v ($w = uv$) s.t. $R_{[u],[v]}$ is true. This ensures that every non-terminal representing a congruence class $[w]$ is capable of deriving each observed string in $[w]$. This constraint is consistent with the definition of SC-CFGs.

Algorithm 2 describes how we can build a formula that encodes these constraints. Line 9 in the algorithm deals with the first and second constraint mentioned above, line 6 with the third constraint and line 8 with the last constraint.

Algorithm 2: Building the Formula

Input: Observation table ⟨K,D,F⟩
Output: A boolean formula, which is made up of a set of clauses

1 $Formula \leftarrow \emptyset$;
2 **foreach** $w \in K$; $|w| > 1$ **do**
3 $\quad RuleVars \leftarrow \emptyset$;
4 $\quad$ **foreach** $uv = w$; $|u|, |v| > 0$ **do**
5 $\quad\quad RuleVars \leftarrow RuleVars \cup \{R_{[u],[v]}\}$;
6 $\quad\quad Formula \leftarrow Formula \cup \{\neg R_{[u],[v]} \vee N_{[u]}, \neg R_{[u],[v]} \vee N_{[v]}\}$;
7 $\quad$ **end**
8 $\quad Formula \leftarrow Formula \cup \{\neg N_{[w]} \vee \bigvee_{r \in RuleVars} r\}$;
9 $\quad$ **if** $(w \in D)$ or $([w] \cap \Sigma \neq \emptyset)$ **then** Set variable $N_{[w]}$ to true ;
10 **end**
11 **return** $Formula$;

As an example, let the following be the congruence classes returned from the previous step:

$1 : [a]$, $2 : [b]$, $3 : [ab, aabb, aaabbb]$, $4 : [aa]$, $5 : [bb]$, $6 : [aab, aaabb]$, $7 : [abb, aabbb]$, $8 : [aaa]$, $9 : [bbb]$, $10 : [aaab]$, $11 : [abbb]$

Then the following is the formula built. Note that the leftmost clauses encode constraint 4, whilst the rest of the clauses encode constraint 3.

Variables N_1 and N_2 are true due to constraint 2, and N_3 is true due to constraint 1.

$\neg N_3 \vee R_{1,2}$	$\neg R_{1,2} \vee N_1$	$\neg R_{1,2} \vee N_2$
$\neg N_3 \vee R_{1,7} \vee R_{4,5} \vee R_{6,2}$	$\neg R_{1,7} \vee N_1$	$\neg R_{1,7} \vee N_7$
$\neg N_3 \vee R_{1,7} \vee R_{4,11} \vee R_{8,9} \vee R_{10,5} \vee R_{6,2}$	$\neg R_{4,5} \vee N_4$	$\neg R_{4,5} \vee N_5$
$\neg N_4 \vee R_{1,1}$	$\neg R_{6,2} \vee N_6$	$\neg R_{6,2} \vee N_2$
$\neg N_5 \vee R_{2,2}$	$\neg R_{4,11} \vee N_4$	$\neg R_{4,11} \vee N_{11}$
$\neg N_6 \vee R_{1,3} \vee R_{4,2}$	$\neg R_{8,9} \vee N_8$	$\neg R_{8,9} \vee N_9$
$\neg N_6 \vee R_{1,3} \vee R_{4,7} \vee R_{8,5} \vee R_{10,2}$	$\neg R_{10,5} \vee N_{10}$	$\neg R_{10,5} \vee N_5$
$\neg N_7 \vee R_{1,5} \vee R_{3,2}$	$\neg R_{1,1} \vee N_1$	
$\neg N_7 \vee R_{1,11} \vee R_{4,9} \vee R_{6,5} \vee R_{3,2}$	$\neg R_{2,2} \vee N_2$	
$\neg N_8 \vee R_{1,4} \vee R_{4,1}$	$\neg R_{1,3} \vee N_1$	$\neg R_{1,3} \vee N_3$
$\neg N_9 \vee R_{2,5} \vee R_{5,2}$	.	.
$\neg N_{10} \vee R_{1,6} \vee R_{4,3} \vee R_{8,2}$	.	.
$\neg N_{11} \vee R_{1,9} \vee R_{3,5} \vee R_{7,2}$	.	.

The grammar built from a solution of the formula consists simply of all the production rules whose rule variables are true and the trivial rules $[a] \to a$ for every $a \in \Sigma$. We shall call *solution grammars* all the grammars built from all the possible solutions for a formula. Note that all solution grammars generate languages which are contained or are equal to the target language because the rules of these grammars are a subset of or equal to the primitive grammar's rules. For example, the following are three grammars built from three different solutions of the example formula (the leftmost and rightmost grammars generate the target language, the middle one generates a subset of the target, $[ab]$ is the starting non-terminal for all grammars):

$[ab] \to [a][abb] \mid [a][b]$	$[ab] \to [aaa][bbb] \mid [aa][bb] \mid [a][b]$	$[ab] \to [aa][abbb] \mid [aa][bb] \mid [a][b]$
$[abb] \to [ab][b]$	$[aaa] \to [a][aa] \quad [bbb] \to [b][bb]$	$[abbb] \to [ab][bb]$
$[a] \to a \quad [b] \to b$	$[aa] \to [a][a] \quad [bb] \to [b][b]$	$[aa] \to [a][a] \quad [bb] \to [b][b]$
	$[a] \to a \quad [b] \to b$	$[a] \to a \quad [b] \to b$

We can use a Min-SAT solver to find an approximate *minimal* solution to the formula. MIN-SAT is the problem of finding the minimal solution that satisfies a given boolean formula, where a minimal solution is one which has the smallest number of true variables. Marathe and Ravi (1996) give a MIN-SAT solver with an approximation ratio of 2. This means that we can find a grammar that is at most twice the size of the smallest grammar in terms of `N+P`. We can get the same result on `N` and `P` by adding negligible weights to rule and non-terminal variables respectively and solve the weighted MIN-SAT problem (Marathe and Ravi, 1996) (which also has an approximation ratio of 2). Note that weighted MIN-SAT is the problem of finding the minimal weighted solution that satisfies a given boolean formula and weights on the variables of this formula, where the minimal weighted solution

is one which minimizes the summation of the weights of the true variables. We can also get the same result on `RHS` and `P+RHS` because `RHS` is always equal to $2\mathtt{P} - |\Sigma|$. However, the main problem we have is that the grammar found might not be able to generate the target language (as is the case in one of the example grammars above).

3.3. Generalizing the Grammar

Algorithm 3 is an attempt to partially solve the problem with the previous step. If we happen to find (through more sampling for example) a string w which is in the target language and not in the learned language, we can use this string to strengthen the boolean formula in such a way that the solution which yielded the incorrect learned grammar would be removed from the space of all possible solutions. In fact, any solution which yields a grammar that does not generate w will also be removed from the solution space.

The idea behind Algorithm 3 is to add additional clauses to the formula so that all solution grammars would have at least one parse tree for w. Since the primitive grammar G' has all the possible parse trees for w, the algorithm simply CYK parses w with G' and encodes the information in the CYK table as clauses.

One clause is added which simply says that at least one production rule that generates the whole string w must be true (line 3 of the algorithm). The rest of the added clauses follow the constraint that if a production rule $[uv] \to [u][v]$ (represented by the rule variable $R_{[u],[v]}$) from the CYK table is chosen, then for every u_1, u_2 s.t. $u_1u_2 = u$ ($|u_1|, |u_2| > 0$), at least one rule $[u] \to [u_1][u_2]$ (represented by the rule variable $R_{[u_1],[u_2]}$) from the CYK table must be chosen. The same applies for v. These constraints are added to the formula in lines 8 and 11 in the algorithm.

Note that we make use of the function `Clause` in Algorithm 3 which simply takes a finite set $\{x_1, x_2 \dots x_n\}$ of literals (i.e. positive or negated variables) and returns a clause $x_1 \vee x_2 \vee \dots \vee x_n$. Also note that $RuleVars$ is simply the set of all positive and negated rule variables for all the production rules in the CYK table.

Algorithm 3 takes polynomial time in the length of w. Note that no new variables are introduced in the formula. Moreover, the number of possible clauses that can be added is finite because everything is bounded by the total number of rules in the primitive grammar. In case all possible clauses are added, the solution grammars will be equivalent to the primitive grammar and thus they will all generate the target language. This means that we only need to call Algorithm 3 on some finite set of strings until every solution grammar is correct. However, the big drawback is that we cannot polynomially bound this set of strings w.r.t. the size of the target grammar or the length of the strings themselves.

Algorithm 3: Strengthening the Formula

Input: A string w in the target language which is not accepted by the learned grammar, the primitive grammar G', the *Formula*

Output: A strengthened *Formula*

$CYKTable \leftarrow CYK(w, G')$;
$RuleVars \leftarrow \{ R_{B,C} \mid A \to BC \in CYKTable \} \cup \{ \neg R_{B,C} \mid A \to BC \in CYKTable \}$;
Add `Clause`$(\{R_{[u],[v]} \mid w = uv;\ |u|, |v| > 0\} \cap RuleVars)$ to *Formula* ;
foreach $x \in Subs(w)$, $|x| > 2$ **do**
  foreach $uv = x$; $|u|, |v| > 0$ **do**
    if $R_{[u],[v]} \notin RuleVars$ **then continue**
    if $|u| > 1$ **then**
      Add `Clause`$(\{\neg R_{[u],[v]}\} \cup \{R_{[u_1],[u_2]} \mid u = u_1u_2;\ |u_1|, |u_2| > 0\} \cap RuleVars)$ to *Formula* ;
    end
    if $|v| > 1$ **then**
      Add `Clause`$(\{\neg R_{[u],[v]}\} \cup \{R_{[v_1],[v_2]} \mid v = v_1v_2;\ |v_1|, |v_2| > 0\} \cap RuleVars)$ to *Formula* ;
    end
  end
end
return *Formula* ;

4. Experiments

We tested our algorithm on 11 typical context-free languages and 9 artificial natural language grammars taken from 4 different sources (Stolcke, 1994; Langley and Stromsten, 2000; Adriaans et al., 2000; Solan et al., 2005). The 11 CFLs include 7 described by unambiguous grammars:

UC1: a^nb^n **UC2**: $a^nb^nc^md^m$ **UC3**: a^nb^m $n \geq m$ **UC4**: a^pb^q, $p \neq q$ **UC5**: Palindromes over alphabet $\{a, b\}$ with a central marker **UC6**: Palindromes over alphabet $\{a, b\}$ without a central marker **UC7**: Lukasiewicz language $(S \to aSS|b)$

and 4 described by ambiguous grammars:

AC1: $|w|_a = |w|_b$ **AC2**: $2|w|_a = |w|_b$ **AC3**: Dyck language **AC4**: Regular expressions.

The 9 artificial natural language grammars are:

NL1:Grammar 'a', Table 2 in (Langley and Stromsten, 2000) **NL2**:Grammar 'b', Table 2 in (Langley and Stromsten, 2000) **NL3**:Lexical categories and constituency, pg 96 in (Stolcke, 1994) **NL4**:Recursive embedding of constituents, pg 97 in (Stolcke, 1994) **NL5**:Agreement, pg 98 in (Stolcke, 1994) **NL6**:Singular/plural NPs and number agreement, pg 99 in (Stolcke, 1994) **NL7**: Experiment 3.1 grammar in (Adriaans et al., 2000) **NL8**: Grammar in Table 10 (Adriaans et al., 2000) **NL9**: TA1 grammar in (Solan et al., 2005).

					Relative Entropy	
Ex.	$\|\Sigma\|$	$\|N\|$	$\|P\|$	$\|S\|$	Our Algorithm	ADIOS
UC1	2	3	4	10	**0.029**	1.876
UC2	4	7	9	50	**0.0**	1.799
UC5	2	3	5	10	**0.111**	7.706
UC7	2	2	3	10	**0.014**	1.257
AC1	2	4	9	50	**0.014**	4.526
AC2	2	5	11	50	**0.098**	6.139
AC3	2	3	5	50	**0.057**	1.934
AC4	7	8	13	100	**0.124**	1.727
NL7	12	3	9	100	**0.0**	0.124
NL1	9	8	15	100	**0.202**	1.646
NL2	8	8	13	200	**0.333**	0.963
NL3	12	10	18	100	**0.227**	1.491
NL5	16	12	23	100	**0.111**	1.692
NL6	19	17	32	400	0.227	**0.138**
UC3	2	3	5	100	**0.411**	0.864
UC4	2	5	9	100	**0.872**	2.480
UC6	2	3	8	100	1.449	**1.0**
NL4	13	11	22	500	**1.886**	2.918
NL8	30	10	35	1000	**1.496**	1.531
NL9	50	45	81	800	1.701	**1.227**

Table 1: Relative Entropy results of our algorithm vs ADIOS (Solan et al., 2005).

Samples where i.i.d. generated from these grammars and given as input to our algorithm. The parameters used for learning the congruence classes were the L1-distance for `dist`, a distance threshold δ which varied considerably depending on the size of the sample (from 0.2 to 0.7), and frequency threshold n roughly the size of the sample divided by 10. The relative entropy[1] between the distributions of the target and learned grammars was calculated on a test set of one million strings generated from the target grammars. The results[2] are shown in Table 1 alongside results obtained by ADIOS (Solan et al., 2005), a system which obtains competitive results on language modelling (Waterfall et al., 2010).

For the tests in the first section of Table 1 (i.e. above the first horizontal line), our algorithm was capable of exactly identifying the structure of the target grammar (after trivial rules with probability 1 in the learned grammar where removed and their RHS replaced in other rules). For the tests in the second section of Table 1 (i.e. between the two horizontal lines), our algorithm was capable of exactly identifying the target non-stochastic language but not the structure of the target grammar. In all of these cases, the induced grammar was slightly smaller than the target one. For the remaining tests, our algorithm

1. The relative entropy (or Kullback-Leibler divergence) between a target distribution D and a hypothesized distribution D' is defined as $\sum_{w\in\Sigma^*} ln\Big(\frac{D(w)}{D'(w)}\Big)D(w)$. Add-one smoothing is used to solve the problem of zero probabilities.
2. The target and learned grammars along with the training samples can be downloaded from https://dl.dropboxusercontent.com/u/3588336/ICGI2014-Dataset.zip

did not identify the target language. In fact, it always overgeneralised. The 3 typical CFLs UC3, UC4 and UC6 are not identified because they are not contained in our subclass of CFLs. Inspite of this, the relative entropy results obtained are still relatively good.

5. Discussion

Clearly there are many things left to analyse about the algorithm presented in this paper. First of all, we do not yet know whether the class of languages generated by SC-CFGs is contained or equal to the class of languages generated by C-CFGs. We conjecture that they coincide. Also, although it is known that C-CFGs in CNF are as expressive as general C-CFGs, we do not know whether SC-CFGs in CNF are as expressive as general SC-CFGs. This is because a binarised version of a SC-CFG might not itself be strongly congruential.

Although we can say something about the merge decisions that need to be taken by Algorithm 1 (see proposition 1), we do not have a clear view of what the non-merge decisions will be in general. What is clear is that the quantity of these decisions can vary with $\prec$.

We feel that a complete theoretical result based on the material in sections 3.2 and 3.3 is possible. One idea is to try to get a stronger type of MAT-learning result where the learned grammar, apart from being equivalent to the target, is also bounded in size w.r.t. the smallest possible grammar. Starting from the learner in (Clark, 2010a), which returns the primitive grammar and the congruence classes, we can find a small grammar through the formula. Using equivalence queries, we can check if this grammar is equivalent to the target. If not equivalent, we can use the returned counter-example string to strengthen the formula using a procedure similar to the one explained in section 3.3 until a correct small grammar is found. As we stated before, the problem with our procedure is that we cannot guarantee that a correct grammar is found after only a polynomial number of calls.

It also seems reasonable to extend the idea of the formula to other classes of grammars which have direct relationships between their non-terminals and observed features of the underlying language (Yoshinaka and Clark, 2010; Kasprzik and Yoshinaka, 2011)

From a more practical point of view, we might have some a priori biases towards certain forms of grammars (apart from smaller grammars). These may be encoded in the formula by giving weights to the variables and finding the minimal weighted solution. For example, a bias can be added in favour of non-terminals representing congruence classes which, according to constituency tests (like the Mutual Information criterion in (Clark, 2001)), are more likely to contain substrings that are constituents.

References

Pieter W. Adriaans, Marten Trautwein, and Marco Vervoort. Towards high speed grammar induction on large text corpora. In Václav Hlavác, Keith G. Jeffery, and Jirí Wiedermann, editors, *SOFSEM*, volume 1963 of *Lecture Notes in Computer Science*, pages 173–186. Springer, 2000.

Wiktor Bartol, Joe Miró, K. Pióro, and Francesc Rosselló. On the coverings by tolerance classes. *Inf. Sci.*, 166(1-4):193–211, 2004.

Luc Boasson and Géraud Sénizergues. NTS Languages Are Deterministic and Congruential. *J. Comput. Syst. Sci.*, 31(3):332–342, 1985.

Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and Abhi Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005.

Alexander Clark. *Unsupervised Language Acquisition: Theory and Practice*. PhD thesis, COGS, University of Sussex, 2001.

Alexander Clark. PAC-Learning Unambiguous NTS Languages. In Yasubumi Sakakibara, Satoshi Kobayashi, Kengo Sato, Tetsuro Nishino, and Etsuji Tomita, editors, *ICGI*, volume 4201 of *Lecture Notes in Computer Science*, pages 59–71. Springer, 2006. ISBN 3-540-45264-8.

Alexander Clark. Distributional learning of some context-free languages with a minimally adequate teacher. In José M. Sempere and Pedro García, editors, *ICGI*, volume 6339 of *Lecture Notes in Computer Science*, pages 24–37. Springer, 2010a.

Alexander Clark. Towards general algorithms for grammatical inference. In Marcus Hutter, Frank Stephan, Vladimir Vovk, and Thomas Zeugmann, editors, *ALT*, volume 6331 of *Lecture Notes in Computer Science*, pages 11–30. Springer, 2010b. ISBN 978-3-642-16107-0.

Alexander Clark. Learning trees from strings: A strong learning algorithm for some context-free grammars. *Journal of Machine Learning Research*, 14:3537–3559, 2013. URL http://jmlr.org/papers/v14/clark13a.html.

Alexander Clark and Rémi Eyraud. Polynomial identification in the limit of substitutable context-free languages. *Journal of Machine Learning Research*, 8:1725–1745, 2007.

Anna Kasprzik and Ryo Yoshinaka. Distributional learning of simple context-free tree grammars. In Jyrki Kivinen, Csaba Szepesvári, Esko Ukkonen, and Thomas Zeugmann, editors, *ALT*, volume 6925 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 2011. ISBN 978-3-642-24411-7.

Pat Langley and Sean Stromsten. Learning context-free grammars with a simplicity bias. In Ramon López de Mántaras and Enric Plaza, editors, *ECML*, volume 1810 of *Lecture Notes in Computer Science*, pages 220–228. Springer, 2000.

K. Lari and S.J. Young. The estimation of stochastic context-free grammars using the inside-outside algorithm. *Computer Speech & Language*, 4(1):35 – 56, 1990.

Madhav V. Marathe and S. S. Ravi. On approximation algorithms for the minimum satisfiability problem. *Inf. Process. Lett.*, 58(1):23–29, 1996.

Wojciech Rytter. Application of lempel-ziv factorization to the approximation of grammar-based compression. *Theor. Comput. Sci.*, 302(1-3):211–222, 2003.

Chihiro Shibata and Ryo Yoshinaka. Pac learning of some subclasses of context-free grammars with basic distributional properties from positive data. In Sanjay Jain, Rémi Munos, Frank Stephan, and Thomas Zeugmann, editors, *ALT*, volume 8139 of *Lecture Notes in Computer Science*, pages 143–157. Springer, 2013. ISBN 978-3-642-40934-9.

Zach Solan, David Horn, Eytan Ruppin, and Shimon Edelman. Unsupervised learning of natural languages. *Proceedings of the National Academy of Sciences of the United States of America*, 102(33):11629–11634, 2005.

Andreas Stolcke. *Bayesian learning of probabilistic language models*. PhD thesis, University of California, Berkeley, 1994.

Heidi R. Waterfall, Ben Sandbank, Luca Onnis, and Shimon Edelman. An empirical generative framework for computational modeling of language acquisition. *Journal of Child Language*, 37:671–703, 6 2010.

C. S. Wetherell. Probabilistic languages: A review and some open questions. *ACM Comput. Surv.*, 12(4):361–379, 1980.

Ryo Yoshinaka. Identification in the limit of k, l-substitutable context-free languages. In Alexander Clark, François Coste, and Laurent Miclet, editors, *ICGI*, volume 5278 of *Lecture Notes in Computer Science*, pages 266–279. Springer, 2008.

Ryo Yoshinaka and Alexander Clark. Polynomial time learning of some multiple context-free languages with a minimally adequate teacher. In Philippe de Groote and Mark-Jan Nederhof, editors, *FG*, volume 7395 of *Lecture Notes in Computer Science*, pages 192–207. Springer, 2010. ISBN 978-3-642-32023-1.