# Automatic Evaluation of Context-Free Grammars (System Description)*

Carles Creus and Guillem Godoy

Universitat Politècnica de Catalunya, Department of Software,
Barcelona, Spain
ccreuslopez@gmail.com, ggodoy@lsi.upc.edu

**Abstract.** We implement an online judge for context-free grammars. Our system contains a list of problems describing formal languages, and asking for grammars generating them. A submitted proposal grammar receives a verdict of acceptance or rejection depending on whether the judge determines that it is equivalent to the reference solution grammar provided by the problem setter. Since equivalence of context-free grammars is an undecidable problem, we consider a maximum length $\ell$ and only test equivalence of the generated languages up to words of length $\ell$. This length restriction is very often sufficient for the well-meant submissions. Since this restricted problem is still NP-complete, we design and implement methods based on hashing, SAT, and automata that perform well in practice.

**Keywords:** grammars, equivalence, hashing, SAT, automata.

## 1 Introduction

Nowadays, there is an increasing interest in offering college-level courses online. Websites like Khan Academy [10], Coursera [13], Udacity [15] and edX [1], provide online courses on numerous topics. The users/students have access to videos and texts explaining several subjects, as well as tools for automated evaluation by means of exercises. In the specific context of computer science, the use of online judges for testing correctness of programs is used in several academic domains as a self-learning tool for students, as well as a precise method in exams for scoring their programming skills (see, e.g., [14,7,2]).

For the last two years we have developed a specific online judge for the subject of Theory of Computation [9], located at `http://racso.lsi.upc.edu/juez`. The site offers exercises about deterministic finite automata, context-free grammars, push-down automata, reductions between undecidable problems, and reductions between NP-complete problems. Users can submit their solutions, the judge evaluates them, and offers a counterexample when the submission is rejected. This is very useful to make students understand why their solutions are wrong, and to keep them motivated during the learning process. We have used

* The authors were supported by an FPU grant (first author) and the FORMALISM project (TIN2007-66523) from the Spanish Ministry of Education and Science.

G. Dowek (ed.): RTA-TLCA 2014, LNCS 8560, pp. 139–148, 2014.

the judge in the classroom, not only as a support tool for the students, but also as an evaluation method on exams. This has had a marked effect on the motivation and involvement of the students: during a fifteen-week course, each student has solved more than 150 problems in average, with more than 680 submissions. This means that each problem needed over 4 submissions to get acceptance from the judge, and the students were motivated enough to perform new attempts to reach an acceptance verdict.

In this paper we explain the techniques used to automatically evaluate the problems on context-free grammars. Each of such problems describes a language $L$ and asks the student to submit a grammar $G_{\texttt{sub}}$ generating $L$. In some cases, it asks specifically for an unambiguous grammar. The judge checks the correctness of $G_{\texttt{sub}}$ by testing that it generates the same language as a reference grammar $G_{\texttt{sol}}$ provided by the problem setter. Since it is well-known that grammar equivalence is an undecidable problem [9], we cannot expect the judge to behave correctly for every input. Therefore, we focus on performing well with the well-meant grammars submitted by students to academic problems. These grammars are very simple, and when they are wrong, there is usually a small counterexample, i.e., a small word in $\mathcal{L}(G_{\texttt{sub}}) \triangle \mathcal{L}(G_{\texttt{sol}})$. For this reason, we tackle the problem by fixing a length $\ell$ and looking for a word $w \in \mathcal{L}(G_{\texttt{sub}}) \triangle \mathcal{L}(G_{\texttt{sol}})$ with length bounded by $\ell$. Since this is still NP-complete, we develop methods that in practice behave well enough for small $\ell$, $G_{\texttt{sub}}$, and $G_{\texttt{sol}}$. In particular, our judges[1] are based on automata and hashing techniques, and we also study possible optimizations for the reduction in [3] of the grammar equivalence to the SAT problem.

The paper is organized as follows. In Section 2 we summarize notations and basic concepts. In Section 3 we explain the different developed methods, and in Section 4 compare them with others from the literature. In Section 5 we describe our online system and our experience using it. We conclude in Section 6.

## 2 Preliminaries

Words are finite-length lists of symbols chosen over an underlying alphabet $\Sigma$. The length of a word $w$ is denoted by $|w|$, and its $i$'th symbol, for $1 \leq i \leq |w|$, is denoted by $w[i]$. Similarly, its subword between $i$ and $j$, inclusive, is denoted by $w[i..j]$. The empty word is denoted by $\varepsilon$. We assume that the reader is familiar with the concept of context-free grammar (CFG) as a structure $G = \langle \mathcal{V}, \Sigma, R, S \rangle$, where $\mathcal{V}$ is the set of non-terminal symbols, $\Sigma$ is the alphabet of terminal symbols, $R \subset \mathcal{V} \times (\mathcal{V} \cup \Sigma)^*$ is the set of rules, and $S \in \mathcal{V}$ is the initial symbol. We denote non-terminals with uppercase letters $X, Y, Z, \ldots$ and terminals with lowercase letters $a, b, c, \ldots$, with possible subscripts. Often, grammars are just represented by a list of rules, where the non-terminal at the left-hand side of the first rule is considered the initial symbol. Also, rules with common left-hand side are usually described in compact form, e.g., two rules $X \to u,\ X \to v$ are represented by $X \to u \mid v$. In order to simplify definitions and arguments, we

[1] Source code available at `http://www.lsi.upc.edu/~ggodoy/publications.html`

assume without loss of generality that the sets of non-terminals of any two grammars are disjoint. We assume that our grammars are reduced and in CNF [9], and hence we only deal with rules of the form $X \to YZ$ and $X \to a$. Recall that the standard transformation to CNF produces a quadratic increase in size, and can be adapted to detect when ambiguity is lost due to the transformation. For a detailed definition of the language $\mathcal{L}(G)$ generated by the CFG $G$, and the concept of ambiguity see, e.g., [9]. We assume that the reader is familiar with the concept of deterministic finite automata (DFA) and their properties [9].

## 3 Judging Methods

### 3.1 Exhaustive

The JudgeExhaustive approach consists in enumerating all the words up to length $\ell$ that can be generated by each of the grammars and checking whether there exists some word $w$ that is generated by just one of them. This brute force solution has some benefits in our setting. First, it is trivial to give the minimal counterexample in size, whenever one exists in our search space. Second, besides enumerating the words, it is easy to count the amount of different derivations that generate each of them. This additional information allows to check whether the grammar is ambiguous in the subset of words with length bounded by $\ell$.

### 3.2 Hash

The JudgeHash approach is based on a hash function $\mathcal{H}$ that maps languages to natural numbers. We focus on the subsets $L_{G_{\mathtt{sub}},\ell} \subseteq \mathcal{L}(G_{\mathtt{sub}})$ and $L_{G_{\mathtt{sol}},\ell} \subseteq \mathcal{L}(G_{\mathtt{sol}})$ of words of length $\ell$ of $\mathcal{L}(G_{\mathtt{sub}})$ and $\mathcal{L}(G_{\mathtt{sol}})$, respectively, and check $L_{G_{\mathtt{sub}},\ell} = L_{G_{\mathtt{sol}},\ell}$ indirectly with $\mathcal{H}(L_{G_{\mathtt{sub}},\ell}) = \mathcal{H}(L_{G_{\mathtt{sol}},\ell})$. Note that by using hash functions we may obtain false positives due to collisions, but never false negatives. We use a typical definition [11] for a hash function for words:

$$\mathcal{H}(w) = \left(\sum_{i=1}^{|w|} w[i] \cdot \mathfrak{b}^{i-1}\right) \bmod \mathfrak{m}$$

where $\mathfrak{m}$ is a "big" prime and $\mathfrak{b}$ is a "small" prime satisfying $\mathfrak{b} > |\Sigma|$. Note that we interpret the terminal symbols in $\Sigma$ as numbers, assuming that they are in $\{1, \ldots, \mathfrak{b}-1\}$ and are pairwise different. An extension of $\mathcal{H}$ to languages like

$$\mathcal{H}(L) = \left(\sum_{w \in L} \mathcal{H}(w)\right) \bmod \mathfrak{m}$$

suffices to detect when $L_{G_{\mathtt{sub}},\ell}$ and $L_{G_{\mathtt{sol}},\ell}$ differ, and in such case a counterexample $w \in L_{G_{\mathtt{sub}},\ell} \triangle L_{G_{\mathtt{sol}},\ell}$ can easily be constructed one symbol at a time: it suffices to check that the symbol appended to $w$ is valid for $w$ to become the counterexample, i.e., to check that $\mathcal{H}(\{wu \in L_{G_{\mathtt{sub}},\ell}\}) \neq \mathcal{H}(\{wv \in L_{G_{\mathtt{sol}},\ell}\})$.

Due to the lack of space and for explanation purposes, instead of giving the formal definition of the efficient computation of $\mathcal{H}(L_{G,\ell})$ making use of the structure of $G$, we just give an example. Consider the language $L = \{a^n b^n \mid n > 0\}$

and the following CFG generating $L$ (already in CNF):

$$\begin{aligned} S &\to AX \mid AB \\ X &\to SB \\ A &\to a \\ B &\to b \end{aligned}$$

By $\mathcal{H}(W,\ell)$ we denote $\mathcal{H}(\{w \in \Sigma^* \mid W \to_G^* w \ \wedge\ |w| = \ell\})$ and by $\mathcal{C}(W,\ell)$ we denote $|\{w \in \Sigma^* \mid W \to_G^* w \wedge |w| = \ell\}|$. Such values can be recursively obtained using the structure of $G$. For the direct cases we have $\mathcal{H}(A,1) = a$, $\mathcal{H}(B,1) = b$, $\mathcal{C}(A,1) = \mathcal{C}(B,1) = 1$, and $\mathcal{H}(A,n) = \mathcal{H}(B,n) = \mathcal{C}(A,n) = \mathcal{C}(B,n) = 0$ for $n > 1$. Since the right-hand sides of rules of $S$ and $X$ have size 2, $\mathcal{H}(S,1) = \mathcal{H}(X,1) = \mathcal{C}(S,1) = \mathcal{C}(X,1) = 0$. Since $X$ only has the rule $X \to SB$, $\mathcal{C}(X,2) = \mathcal{C}(S,1) \cdot \mathcal{C}(B,1) = 0$, and $\mathcal{H}(X,2) = \mathcal{H}(S,1) \cdot \mathcal{C}(B,1) + \mathcal{C}(S,1) \cdot \mathcal{H}(B,1) \cdot \mathfrak{b} = 0$. Proceeding analogously, we obtain $\mathcal{C}(S,2) = 1,\ \mathcal{H}(S,2) = a + b\mathfrak{b}$. Since $S$ has the rules $S \to AX,\ S \to AB$,

$$\begin{aligned} \mathcal{C}(S,3) \ &= \mathcal{C}(A,1) \cdot \mathcal{C}(X,2) + \mathcal{C}(A,2) \cdot \mathcal{C}(X,1) + \\ &\quad\ \mathcal{C}(A,1) \cdot \mathcal{C}(B,2) + \mathcal{C}(A,2) \cdot \mathcal{C}(B,1) = 0 \\ \mathcal{H}(S,3) &= \mathcal{H}(A,1) \cdot \mathcal{C}(X,2) + \mathcal{C}(A,1) \cdot \mathcal{H}(X,2) \cdot \mathfrak{b} + \\ &\quad\ \mathcal{H}(A,2) \cdot \mathcal{C}(X,1) + \mathcal{C}(A,2) \cdot \mathcal{H}(X,1) \cdot \mathfrak{b}^2 + \\ &\quad\ \mathcal{H}(A,1) \cdot \mathcal{C}(B,2) + \mathcal{C}(A,1) \cdot \mathcal{H}(B,2) \cdot \mathfrak{b} + \\ &\quad\ \mathcal{H}(A,2) \cdot \mathcal{C}(B,1) + \mathcal{C}(A,2) \cdot \mathcal{H}(B,1) \cdot \mathfrak{b}^2 = 0 \end{aligned}$$

Proceeding analogously, we obtain $\mathcal{C}(X,3) = 1,\ \mathcal{H}(X,3) = a + b\mathfrak{b} + b\mathfrak{b}^2$.

JudgeHash only works correctly when $G_{\texttt{sol}}$ is unambiguous because derivations generating the same word are counted independently. The generated counterexample $w$ to the correctness of $G_{\texttt{sub}}$ will be either a word in $\mathcal{L}(G_{\texttt{sub}}) \triangle \mathcal{L}(G_{\texttt{sol}})$ or a word ambiguously generated by $G_{\texttt{sub}}$. A membership test can determine which one of these cases takes place.

### 3.3 SAT

The JudgeSAT is based on the work of [3] and consists in testing equivalence of $G_{\texttt{sub}}$ and $G_{\texttt{sol}}$ by reducing the problem to the satisfiability of boolean propositional formulas. More specifically, the idea is to first construct a formula $\mathcal{F}_{\ell,G_{\texttt{sub}},G_{\texttt{sol}}}$ such that it is satisfiable if and only if there exists a counterexample word of length at most $\ell$, and then to solve the formula with a state-of-the-art SAT solver. We have reimplemented this method with the idea of trying some possible optimizations. One of them consists in splitting $\mathcal{F}_{\ell,G_{\texttt{sub}},G_{\texttt{sol}}}$ in two independent formulas $\mathcal{F}_{\ell,G_{\texttt{sub}}\setminus G_{\texttt{sol}}}$ and $\mathcal{F}_{\ell,G_{\texttt{sol}}\setminus G_{\texttt{sub}}}$, where $\mathcal{F}_{\ell,G_i\setminus G_j}$ is satisfiable if and only if there exists a word of length $\ell$ in $\mathcal{L}(G_i) \setminus \mathcal{L}(G_j)$. We recall the reduction process of [3] just for $\mathcal{F}_{\ell,G_{\texttt{sub}}\setminus G_{\texttt{sol}}}$.

The formula $\mathcal{F}_{\ell,G_{\texttt{sub}}\setminus G_{\texttt{sol}}}$ is defined by means of two kinds of propositional variables: $\mathcal{X}_i^a$ and $\mathcal{X}_{i,j}^X$, where $1 \le i \le j \le \ell$, $a \in \Sigma$ and $X \in \mathcal{V}$. The first kind of variable, $\mathcal{X}_i^a$, represents the fact that the counterexample $w$ has the terminal $a$ at position $i$. The second kind of variable, $\mathcal{X}_{i,j}^X$, represents the fact

that the subword $w[i..j]$ can be generated from the non-terminal $X$. The formula $\mathcal{F}_{\ell,G_{\texttt{sub}}\setminus G_{\texttt{sol}}}$ can be decomposed into four different parts. First, it guarantees that the counterexample $w[1..\ell]$ is a valid word in $\Sigma^*$ by forcing each position of the word to contain exactly one terminal symbol of $\Sigma$:

$$\bigwedge_{i=1}^{\ell}\Big(\bigvee_{a\in\Sigma}\mathcal{X}_i^a\Big)$$
$$\bigwedge_{i=1}^{\ell}\bigwedge_{a\in\Sigma}\Big(\mathcal{X}_i^a\rightarrow\bigwedge_{b\in\Sigma\setminus\{a\}}\neg\mathcal{X}_i^b\Big)$$

Second, $\mathcal{F}_{\ell,G_{\texttt{sub}}\setminus G_{\texttt{sol}}}$ states that $w[1..\ell]$ is generated by $G_{\texttt{sub}}$ but not by $G_{\texttt{sol}}$ with the two unit clauses $(\mathcal{X}_{1,\ell}^{S_{\texttt{sub}}})$ and $(\neg\mathcal{X}_{1,\ell}^{S_{\texttt{sol}}})$, where $S_{\texttt{sub}}$ and $S_{\texttt{sol}}$ are the starting symbols of $G_{\texttt{sub}}$ and $G_{\texttt{sol}}$, respectively. Third, it formalizes the fact that $G_{\texttt{sub}}$ generates $w[1..\ell]$:

$$\bigwedge_{i=1}^{\ell-1}\bigwedge_{j=i+1}^{\ell}\bigwedge_{X\in\mathcal{V}_{\texttt{sub}}}\Big(\mathcal{X}_{i,j}^X\rightarrow\bigvee_{(X\rightarrow YZ)\in R_{\texttt{sub}}}\bigvee_{s=i}^{j-1}(\mathcal{X}_{i,s}^Y\wedge\mathcal{X}_{s+1,j}^Z)\Big)$$
$$\bigwedge_{i=1}^{\ell}\bigwedge_{X\in\mathcal{V}_{\texttt{sub}}}\Big(\mathcal{X}_{i,i}^X\rightarrow\bigvee_{(X\rightarrow a)\in R_{\texttt{sub}}}\mathcal{X}_i^a\Big)$$

where $\mathcal{V}_{\texttt{sub}}$ and $R_{\texttt{sub}}$ are the sets of non-terminals and rules of $G_{\texttt{sub}}$. And fourth, it formalizes the fact that $G_{\texttt{sol}}$ does not generate $w[1..\ell]$ with a formula analogous to the previous one, but with the direction of the implications reversed.

It is clear that a counterexample $w$ of length $\ell$ exists if and only if either $\mathcal{F}_{\ell,G_{\texttt{sub}}\setminus G_{\texttt{sol}}}$ or $\mathcal{F}_{\ell,G_{\texttt{sol}}\setminus G_{\texttt{sub}}}$ is satisfiable. Moreover, $w$ can be derived from any assignment $\eta$ that satisfies a formula by analysing all the values $\eta(\mathcal{X}_i^a)$.

One additional optimization with respect to [3] that we have considered is to simplify the formulas as follows: whenever we detect that a non-terminal $X$ cannot generate any word of length $k$, we simply create a unit clause $(\neg\mathcal{X}_{i,i+k-1}^X)$ for any relevant $i$. This allows us to ignore for length $k$ all the rules with $X$ as left-hand side and all the possible split indexes $s\in\{i,\ldots,i+k-1\}$.

### 3.4 DFA

The approach of JudgeDFA is based on automata techniques. The idea is to construct the minimum DFA $A_{\texttt{sub},\ell}$ and $A_{\texttt{sol},\ell}$ recognizing the words of length $\ell$ generated by $G_{\texttt{sub}}$ and $G_{\texttt{sol}}$, and testing whether $A_{\texttt{sub},\ell}$ and $A_{\texttt{sol},\ell}$ are identical. In order to be able to compute the automata directly on the grammars, we use the following function $\mathcal{A}:\mathcal{V}\times\mathbb{N}\rightarrow\text{DFA}$ mapping a non-terminal $X$ and a length $\ell$ to an automaton recognizing the words of length $\ell$ generated from $X$:

$$\mathcal{A}(X,\ell)=\begin{cases}\bigcup_{(X\rightarrow a)\in R}A_a & \text{if } \ell{=}1\\ \bigcup_{(X\rightarrow YZ)\in R}\bigcup_{i=1}^{\ell-1}\mathcal{A}(Y,i)\cdot\mathcal{A}(Z,\ell-i) & \text{if } \ell{>}1\end{cases}$$

where concatenation and union are not set operations, but automata operations producing automata recognizing the concatenation and union of the languages of the given automata, and $A_a$ denotes the automaton recognizing the word $a\in\Sigma$. Moreover, we assume that the DFA recognizing the empty language is the neutral element of the automata union. Note that we do not explicitly detect whether the grammars are ambiguous, but an approximation could be checked while evaluating $\mathcal{A}$ by testing whether the unions performed are disjoint.

# 4 Performance

## 4.1 Complexity Analysis

For JudgeExhaustive, when we restrict to small alphabets and small $\ell$, words can be encoded as natural numbers using the native representation of the computer, and then the running time is in $\mathcal{O}(|R|\cdot\ell\cdot|\Sigma|^\ell)$ and its space in $\mathcal{O}(\mathcal{V}\cdot|\Sigma|^\ell)$.

In the case of JudgeHash, using a dynamic programming scheme, the space requirements to compute $\mathcal{C}$ and $\mathcal{H}$ up to the current counterexample $w$ are in $\mathcal{O}(|\mathcal{V}|\cdot\ell^2)$ and it takes time in $\mathcal{O}(|R|\cdot\ell^3)$. The construction of the counterexample $w$ requires the recomputation of $\mathcal{C}_w$ and $\mathcal{H}_w$ at most $\ell\cdot|\Sigma|$ times, giving a global running time in $\mathcal{O}(|R|\cdot\ell^4\cdot|\Sigma|)$.

The number of variables of the form $\mathcal{X}_i^a$ and $\mathcal{X}_{i,j}^X$ of JudgeSAT is in $\mathcal{O}(\ell\cdot|\Sigma|+\ell^2\cdot|\mathcal{V}|)$, and hence, the cost of solving $\mathcal{F}_{\ell,G_i\setminus G_j}$ is in $2^{\mathcal{O}(\ell\cdot|\Sigma|+\ell^2\cdot|\mathcal{V}|)}$. Fortunately, state-of-the-art SAT solvers perform much better than this in practice, and incremental SAT-solver techniques [8,16] lead to noticeable speed-ups since the solver can reuse for $\mathcal{F}_{\ell+1,G_i\setminus G_j}$ the knowledge obtained when solving $\mathcal{F}_{\ell,G_i\setminus G_j}$.

Finally, for JudgeDFA, first note that the size of a minimum DFA recognizing a set of words of length $\ell$ is in $2^{\mathcal{O}(\log(|\Sigma|)\cdot\ell)}$. Second, recall that the concatenation operation on automata might lead to exponential blowup and the union operation multiplies the sizes of the automata being considered [9]. This implies that the computations of JudgeDFA have space requirements in $2^{2^{\mathcal{O}(\log(|\Sigma|)\cdot\ell)}}$. However, this extreme bound is only reached when the languages generated by the grammars require excessive memorization, e.g., the language of palindromes and the language of well-parenthesized words. In other cases, JudgeDFA can "compress" the language by sharing states and obtains much better performance.

## 4.2 Benchmarks

We have compared the performance of our judges using the set of benchmarks from [3], comprising 35910 different pairs of grammars. As a reference, we have also tested the prototype implementation of [3], `cfganalyzer`[2]. Additionally, we have also compared our proposals with Hampi[3], a state-of-the-art string-constraint solver developed in [6] which is expressive enough to approximate grammar equivalence. Hampi works by internally transforming the grammars to fixed-length regular expressions and then solving the constraints for them.

Figure 1 shows the results obtained[4] for each judge when testing grammar equivalence up to length $\ell = 15$. The plot can be interpreted as follows: all the 35910 tests are run in parallel in independent machines, the abscissa is the elapsed time, and the ordinate is the number of tests that have not reached a verdict. It is easy to see in the chart that most of the tests can be solved in less

[2] Version 2012-12-26, `http://www2.tcs.ifi.lmu.de/~mlange/cfganalyzer/`

[3] Version 2012-2-13, `http://people.csail.mit.edu/akiezun/hampi/`

[4] Measurements taken on a 64-bit Intel® Pentium® T4200, at 2GHz and with 4GB of RAM, timings available at `http://www.lsi.upc.edu/~ggodoy/publications.html`

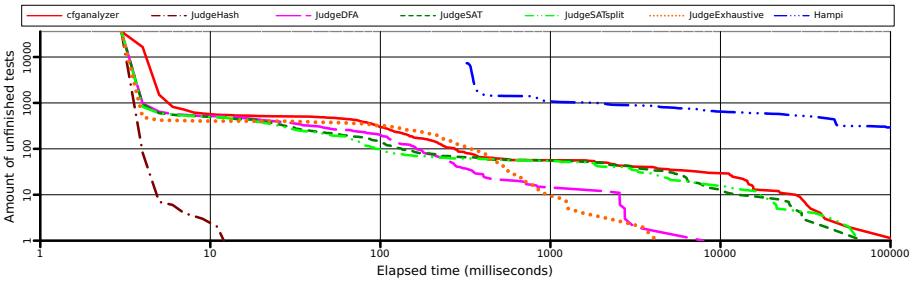



**Fig. 1.** Results using the benchmarks from [3], comprising 35910 grammar pairs, testing equivalence up to length $\ell = 15$

than 10 milliseconds by all the judges. This is because in such cases either the counterexamples were small, i.e., the empty word, or the grammars were deemed wrongly formatted. Timings for `cfganalyzer` and JudgeSAT correspond to the builds using the latest version of MiniSat[5] [5] (the solvers zChaff [12] and PicoSat [4] were also considered, but finally discarded due to lower performance). Clearly, the plots for JudgeSAT closely follow `cfganalyzer`, with our optimizations giving only a small benefit. The results for JudgeExhaustive and JudgeDFA are comparable to `cfganalyzer`, and even significantly better when considering the most expensive tests. JudgeExhaustive is competitive with the rest because the alphabets are small and the languages are sparse. The timings obtained for JudgeHash are quite remarkable: the worst running time was just 12 milliseconds. Finally, in the case of Hampi, we had to limit the execution time to 2 minutes and the memory to 512 MB, since in some tests the program hung consuming all the available memory. Due to these problems, we only used a subset of the tests. Overall, results for Hampi are rather poor when compared to the rest of judges.

### 4.3 Stressing the Judges

To highlight the differences between our judges, we have devised an additional set of grammar pairs that focus on particular bottlenecks of each judge:

- Language of words with same number of `a`'s and `b`'s:

$$G_{\texttt{numab}} = \{S \to \texttt{a}X\texttt{b}S \mid \texttt{b}Y\texttt{a}S \mid \varepsilon,\quad X \to \texttt{a}X\texttt{b}X \mid \varepsilon,\quad Y \to \texttt{b}Y\texttt{a}Y \mid \varepsilon\} \qquad G'_{\texttt{numab}} = \{S \to \texttt{a}S\texttt{b}S \mid \texttt{b}S\texttt{a}S \mid \varepsilon\}$$

- Palindromes over an alphabet $\Sigma_i$ with $i$ different terminal symbols:

$$G_{\texttt{pal}_i} = \bigcup_{\texttt{a}\in\Sigma_i}\{S \to \texttt{a}S\texttt{a} \mid \texttt{a} \mid \varepsilon\} \qquad G'_{\texttt{pal}_i} = \bigcup_{\texttt{a}\in\Sigma_i}\{S \to A_{\texttt{a}} \mid \texttt{a} \mid \varepsilon,\ A_{\texttt{a}} \to \texttt{a}S\texttt{a}\}$$

[5] Version 2.2.0, `http://www.minisat.se`

– Language of the well-parenthesized words:

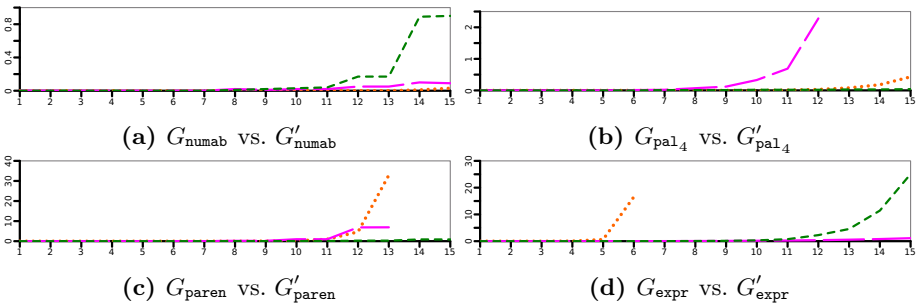
$$G_{\texttt{paren}} = \{S \to \texttt{(}S\texttt{)}S \mid \texttt{[}S\texttt{]}S \mid \{S\}S \mid \texttt{<}S\texttt{>}S \mid \varepsilon\} \qquad G'_{\texttt{paren}} = \{S \to S\texttt{(}S\texttt{)}S \mid S\texttt{[}S\texttt{]}S \mid S\{S\}S \mid S\texttt{<}S\texttt{>}S \mid \varepsilon\}$$

– Language of the valid expressions over the alphabet $\Sigma = \{\texttt{+}, \texttt{*}, \texttt{(}, \texttt{)}, \texttt{0}, \ldots, \texttt{9}\}$:

$$G_{\texttt{expr}} = \{E \to P\texttt{+}E \mid P,\ P \to B\texttt{*}P \mid B,\ B \to N \mid \texttt{(}E\texttt{)},\ N \to ND \mid D,\ D \to \texttt{0} \mid \texttt{1} \mid \ldots \mid \texttt{9}\}$$

$$G'_{\texttt{expr}} = \{E \to E\texttt{+}E \mid E\texttt{*}E \mid N \mid \texttt{(}E\texttt{)},\ N \to DN \mid D,\ D \to \texttt{0} \mid \texttt{1} \mid \ldots \mid \texttt{9}\}$$

Figure 2 shows the obtained running times. As expected, JudgeExhaustive achieves acceptable performance only when the alphabet is small and the languages sparse. The cases of JudgeDFA and JudgeSAT are related: the latter improves the times of the former when the languages require excessive memorization. For instance, $G_{\texttt{pal}_i}$, $G'_{\texttt{pal}_i}$, $G_{\texttt{paren}}$ and $G'_{\texttt{paren}}$ force the automata to memorize almost all the read word. When no such excessive memorization is required, like in $G_{\texttt{numab}}$, $G'_{\texttt{numab}}$, $G_{\texttt{expr}}$ and $G'_{\texttt{expr}}$, the automata of JudgeDFA are small and it obtains better performance than JudgeSAT.



**(a)** $G_{\texttt{numab}}$ vs. $G'_{\texttt{numab}}$ **(b)** $G_{\texttt{pal}_4}$ vs. $G'_{\texttt{pal}_4}$

**(c)** $G_{\texttt{paren}}$ vs. $G'_{\texttt{paren}}$ **(d)** $G_{\texttt{expr}}$ vs. $G'_{\texttt{expr}}$

**Fig. 2.** Running times in seconds (ordinate) in terms of the maximum length $\ell$ tested (abscissa) for the different judges (dotted lines correspond to JudgeExhaustive, long-dashed to JudgeDFA, short-dashed to JudgeSAT)

We have an extra test to stress JudgeHash. This is necessary since its worst case scenario is not with equivalent grammars, but when there is a big counterexample. We test the language of words of length $i$ over $\Sigma_j = \{\texttt{a}_1, \ldots, \texttt{a}_j\}$:

$$G_{\texttt{art}_{i,j}} = \{S \to \overbrace{T \ldots T}^{i},\ T \to \texttt{a}_1 \mid \ldots \mid \texttt{a}_j\} \qquad G'_{\texttt{art}_{i,j}} = G_{\texttt{art}_{i,j}} \cup \{S \to \overbrace{\texttt{a}_j \ldots \texttt{a}_j}^{i}\}$$

Note that the lexicographically last word is ambiguously generated by $G'_{\texttt{art}_{i,j}}$, thus being the counterexample. Figure 3 depicts the obtained times, where it is clear that competitive performance is achieved even with rather big $i$ and $j$.

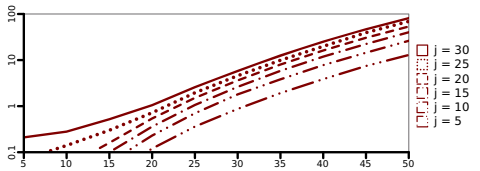



**Fig. 3.** JudgeHash's running times in seconds (ordinate) in terms of the word length $i$ (abscissa) for $G_{\mathtt{art}_{i,j}}$ vs. $G'_{\mathtt{art}_{i,j}}$

## 5 Online Judging System

Our website currently offers 46 problems on CFG's and 21 on push-down automata, which are checked by the same judges by a prior standard transformation into CFG's. Our system is configured as follows. For problems with languages over alphabets with 2 or 3 symbols we use JudgeExhaustive with $\ell = 10$, because it is fast enough, has a rather uniform running time, and the answer is guaranteed to be correct up to the chosen $\ell$. For problems over alphabets larger than 3 and asking for an unambiguous grammar we use JudgeHash and $\ell = 15$, which is combined with JudgeExhaustive with $\ell = 3$ in order to reduce the chances of hash collisions. For the rest of problems we use either JudgeSAT or JudgeDFA, depending on the language, and with $\ell = 10$. Essentially, JudgeDFA is used for those languages for which the expected natural grammar solutions produce small automata, according to the problem setter criterion.

We have been using the online judge since September 2012 with the students of the Theory of Computation subject at the Computer Science course of the Universitat Politècnica de Catalunya. For the first two semesters it was offered to the students as an optional support tool to do exercises. During the fall semester of 2013 we also used our system to hold online exams, and the students made over 12000 submissions in total, for an average of 250 submissions per student.

## 6 Conclusions

We have developed several techniques for determining if two given context-free grammars generate the same language. The methods we have implemented work sufficiently well in practice. In the case of the SAT-based judge, the performance of our implementation is similar to the state of the art. The hash-based method has much better performance than the others. Nevertheless, besides the fact that this method cannot be used with ambiguous solution grammars, the extension of the hash function from words to languages degrades some of its properties, and it may happen that some collisions take place independently of the chosen primes $\mathfrak{m}, \mathfrak{b}$. For instance, the following languages $L_1, L_2$ give rise to the same value through the hash:

$$L_1 = \{a^n b^n \mid n \geq 0\} \cup \{c^n d^n \mid n \geq 0\}$$
$$L_2 = \{a^n d^n \mid n \geq 0\} \cup \{c^n b^n \mid n \geq 0\}$$

This problem takes place only in specific languages in practice, but defining alternative hashing functions to avoid it should be matter of further work.

According to the students' opinions, the judge is a good support tool that helps them know if they are understanding the matter. In our opinion, it has all the benefits of online judges: it is a good support-learning tool, gives instant feedback, and motivates users to practice. Note that the tool just checks that the submitted solutions are correct, but neither the quality of such solutions nor that the students have understood them well enough to justify their correctness. In this sense, the professor is essential to adquire a good comprehension of the matter.

Although many problems in the list are artificial, they help students to understand the limits of expressivity of CFG's, and how context-free conditions can be combined with regular conditions. We are interested in studying whether similar techniques perform well for evaluating grammars designed for descending or ascending parsing, and for the construction of abstract syntax trees. This can be useful to develop support-learning tools for the Compilers subject.

## References

1. Agarwal, A.: edX (2012), `https://www.edx.org`
2. Alexandrova, S., Balandin, A., Compeau, P., Kladov, A., Rayko, M., Sosa, E., Vyahhi, N., Dvorkin, M.: Rosalind (2012), `http://www.rosalind.info`
3. Axelsson, R., Heljanko, K., Lange, M.: Analyzing context-free grammars using an incremental SAT solver. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part II. LNCS, vol. 5126, pp. 410–422. Springer, Heidelberg (2008)
4. Biere, A.: PicoSAT essentials. Journal on Satisfiability, Boolean Modeling and Computation 4(2-4), 75–97 (2008)
5. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
6. Ganesh, V., Kieżun, A., Artzi, S., Guo, P.J., Hooimeijer, P., Ernst, M.: HAMPI: A string solver for testing, analysis and vulnerability detection. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 1–19. Springer, Heidelberg (2011)
7. García, C., Revilla, M.A.: UVa online judge (1997), `http://uva.onlinejudge.org`
8. Hooker, J.N.: Solving the incremental satisfiability problem. Journal of Logic Programming 15(1&2), 177–186 (1993)
9. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation, 3rd edn. Addison-Wesley (2006)
10. Khan, S.: Khan Academy (2006), `https://www.khanacademy.org`
11. Knuth, D.E.: The Art of Computer Programming, vol. III: Sorting and Searching. Addison-Wesley (1973)
12. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Annual ACM IEEE Design Automation Conference, pp. 530–535. ACM (2001)
13. Ng, A., Koller, D.: Coursera (2012), `https://www.coursera.org`
14. Petit, J., Giménez, O., Roura, S.: Jutge.org: an educational programming judge. In: ACM Special Interest Group on Computer Science Education, pp. 445–450 (2012)
15. Thrun, S., Stavens, D., Sokolsky, M.: Udacity (2012), `https://www.udacity.com`
16. Whittemore, J., Kim, J., Sakallah, K.A.: SATIRE: A new incremental satisfiability engine. In: Design Automation Conference, pp. 542–545 (2001)