A JFLAP Extension for Checking Context-Free Grammars

Priyanka Bansal and Viraj Kumar
1Computer Science and Engineering, NIIT University, Neemrana, India
2Department of Information Science and Engineering, P.E.S. Institute of Technology, Bangalore, India

Abstract - JFLAP is a freely available and popular software package used in formal languages and automata courses. We develop a JFLAP extension for checking homework assignments involving the design of context-free grammars (CFGs). Determining the language equivalence of arbitrary CFGs is undecidable. As a result, instructors as well as students typically resort to manual checking that is slow and error-prone. For instructors, we automate this process by verifying that student CFGs match the instructor’s CFG for all strings of length up to L (an instructor-supplied parameter). Our tool is easy to use and is practical for large batches of students. It can also be used by students (e.g., to test whether two approaches to a particular problem are equivalent up to length L), giving valuable feedback quickly and reliably.

Keywords: Computer-aided assessment, Formal Languages, Context-Free Grammars, JFLAP.

1 Introduction

Homework assignments in formal languages and automata (FLA) courses are often paper-and-pen based. They typically include problems of constructing automata or grammars for a given formal language. Testing (for students) and grading (for instructors) is usually a laborious and error-prone process. The advent and adoption of tools such as JFLAP [1] and L-FLAT [2] is, however, changing this landscape. Students can use such software to code solutions, and easily design and run complex unit-tests. Failing such tests gives students valuable feedback and enables them to correct their solutions, whereas passing such tests gives confidence in the correctness of their approach. JFLAP is extremely popular world-wide (see [3] for usage statistics) because it covers nearly every topic in the FLA course. Furthermore, the community actively encourages the development of grading support for instructors [4]. The present paper is similar to [5] in its contribution: whereas [5] is a tool for checking equivalence of finite automata, our tool focuses on context-free grammars (CFGs).

It is well known that CFG language equivalence is an undecidable problem [6]. Most instructors, therefore, apply a heuristic similar to this: If there is a short counter-example x, then mark INCORRECT, else if the student CFG “looks OK” then mark CORRECT, else mark as per the grading policy.

A “counter-example” is any string x that is generated by one of the CFGs, but not the other. One can easily automate the search for such a string up to a finite, instructor-specified length L. On the other hand, it is not clear what it means for a CFG to “look OK”. Indeed, this subjective judgment is where errors and inconsistencies in grading are most likely to creep in. We can eliminate this imprecise test to obtain the following modified grading heuristic: If there is a counter-example of length ≤ L then mark INCORRECT, else mark CORRECT.

A student solution that simply generates the requisite strings up to length L will, of course, be deemed “correct” under our heuristic. However, students are unlikely to attempt such tailored solutions for at least two reasons: (1) the instructor need not reveal L, and (2) if L is large enough, the tailored CFG is likely to be conspicuously larger (in terms of variables and rules) than an honest attempt.

Computer-aided assessment tools such as ours offer a reliable and effective way of grading computer science assignments, with a history of success spanning over five decades [7] (see [8] for a detailed overview). In addition to speed and accuracy, automation has two further benefits, both of which are particularly relevant in FLA courses: (1) By streamlining the grading process, our tool encourages instructors to assign more problems than time-constraints would allow under manual grading, and (2) our tool is also available to learners, providing automated feedback which is not prone to the errors and delays of manual feedback. This encourages learners to experiment with a variety of approaches to problem-solving, which is a prized skill in this domain.

In order to leverage these benefits, we have tried to make our tool easy to install and immediately usable. (A much more sophisticated and faster prototype application is available [9], but this is not designed as a computer-aided assessment tool and is therefore harder to use.) Also, to keep our tool as general as possible, we do not assume that the CFGs in question have efficient parsers. Our tool does, at present, have two primary limitations: (1) memory and running-time constraints limit L to about 15, and (2) there

1 Since there are exponentially many such strings, some instructors test only a small random sample of such strings.
cannot be more than 64 variables in our internal Chomsky Normal Form (CNF) representation of the given CFGs \(^2\). Both these limitations are the consequence of our rather naïve initial implementation, and are being addressed.

The remainder of the paper is structured as follows. In Section 2 we explain how our tool integrates with JFLAP, and demonstrate features that are helpful to both students and instructors. Section 3 details experiments to determine the practicality of our tool. Finally, in Section 4 we conclude with our future research directions. The latest version of our tool is available at www.niituniversity.in/projects/CFGeq.zip, with a README file describing the installation steps.

2 Integration with JFLAP and usage

Our tool integrates with JFLAP via an intuitive access-point under the Test tab for Grammars. We first demonstrate the simplest usage of our tool, Test for Equivalence (Fig. 1). The user selects two CFGs and a length bound \(L\), and the tool compares the CFGs on all strings of length \(\leq L\).

![Figure 1. Testing two CFGs for equivalence](image1.png)

The CFGs in Fig. 1 are obviously equivalent (the only difference being that variable \(A\) in the first CFG has been renamed \(D\) in the second CFG), and the tool reports the expected result (Fig. 2).

![Figure 2. CFGs equivalent up to tested length \(L = 10\)](image2.png)

This simple functionality can nevertheless be extremely valuable for learners. Consider the somewhat more complex example in Fig. 3 where a student is attempting to remove so-called “unit rules” \((S \rightarrow A\) and \(S \rightarrow B)\) from the left-hand CFG to obtain an equivalent CFG without such rules. In following a well-known algorithm for this, our student has made a small mistake in the CFG on the right.

![Figure 3. Two inequivalent CFGs](image3.png)

Our tool reports that the CFGs are inequivalent, and also displays a shortest counter-example establishing this fact \(^3\). Our student may now guess that one extra rule \((S \rightarrow 1)\) fixes the problem, and she can quickly test this hypothesis (Fig. 4).

![Figure 4. Correcting the error by adding one more rule](image4.png)

---

\(^2\) JFLAP restricts CFG variable names to the 26 uppercase letters of the alphabet, so ours is not a severe limitation. Also, although JFLAP provides an automatic conversion of CFGs into CNF, it uses an inefficient process. Our internal CNF representation uses the more efficient process detailed in [10].

\(^3\) As per Occam’s principle, a shortest counter-example is likely to be most informative.
We believe our hypothetical student can greatly benefit from this kind of clear and rapid feedback. Having found the fix, she can easily discover that her earlier mistake was (in this case) caused by improper book-keeping while executing the algorithm, and not by a fundamental misunderstanding. In the latter case, she would probably need to devote significant time to properly understand the process, whereas here she knows that she merely needs to be watchful.

Instructors are likely to find the second feature of our tool more helpful: **Batch Test for Equivalence** (Fig. 5).

![Image of Batch Test for Equivalence](image)

Figure 5. Batch testing: instructor CFG and student submissions folder

To perform a batch test, the instructor first opens the sample solution and clicks **Batch Test for Equivalence**. The tool then asks the instructor to choose a folder containing student submissions and the bounded length $L$. The tool checks each student’s submission against the instructor’s sample solution on strings of length $\leq L$, and reports the results sequentially using dialog boxes similar to Fig. 2 and Fig. 3.

### 3 Experiments

In this section, we report running times for our tool on “difficult” inputs. We focus only on the **Test for Equivalence**, since batch testing merely performs this equivalence test on multiple inputs.

If both CFGs are defined over an alphabet with $m$ letters, there are $\Theta(m^L)$ strings of length at most $L$. The most naïve approach would be to examine each such string $x$ and determine whether both CFGs can derive $x$, or neither can, using the classical CYK algorithm [11][12][13]. This would yield a worst-case running time of $O(rL^3m^L)$, where $r$ is the number of rules in the CNF grammar. The current version of our tool uses a modified CYK algorithm that remembers the derivations of short strings (in an $m$-ary tree data structure), and can reference this information while parsing longer strings. A careful implementation yields a worst-case running time of $O(rL^2m^L)$, which is about 50−100 times faster than the naïve approach on typical inputs.

For our experiments, we generate random CFGs (in CNF) with $m$ letters and $v$ variables, and set $r = mv$ (this choice tends to generate “hard” test cases). For each such CFG, we generate a new CFG by randomly permuting the names of the variables (except the start-variable). The new CFG is clearly equivalent to the original, and forces our code to run for the longest possible time. For such adversarial inputs, we report the run-times (in milliseconds) obtained by running our code on a low-end Windows XP version 5.1 machine, with a Pentium® Dual-Core 2.6 GHz CPU and 2 GB RAM. The data in Tables 1 to 4 below is averaged over four runs.

<table>
<thead>
<tr>
<th>$L$</th>
<th>8</th>
<th>10</th>
<th>12</th>
<th>14</th>
<th>16</th>
<th>18</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>16</td>
<td>78</td>
<td>344</td>
<td>1719</td>
<td>7766</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>4</td>
<td>24</td>
<td>98</td>
<td>450</td>
<td>2164</td>
<td>9270</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>8</td>
<td>28</td>
<td>141</td>
<td>471</td>
<td>3231</td>
<td>14848</td>
</tr>
<tr>
<td>15</td>
<td>4</td>
<td>8</td>
<td>39</td>
<td>180</td>
<td>887</td>
<td>5048</td>
<td>18062</td>
</tr>
<tr>
<td>20</td>
<td>4</td>
<td>12</td>
<td>55</td>
<td>235</td>
<td>1142</td>
<td>5891</td>
<td>23676</td>
</tr>
</tbody>
</table>

Table 1. Run-time in milliseconds ($m = 2, r = mv$)

<table>
<thead>
<tr>
<th>$L$</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td>16</td>
<td>43</td>
<td>145</td>
<td>405</td>
<td>1636</td>
<td>5723</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>16</td>
<td>59</td>
<td>200</td>
<td>735</td>
<td>2230</td>
<td>8012</td>
</tr>
<tr>
<td>10</td>
<td>12</td>
<td>28</td>
<td>117</td>
<td>371</td>
<td>1274</td>
<td>4184</td>
<td>13223</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>43</td>
<td>149</td>
<td>515</td>
<td>1684</td>
<td>6524</td>
<td>18265</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>75</td>
<td>227</td>
<td>829</td>
<td>2180</td>
<td>7656</td>
<td>33375</td>
</tr>
</tbody>
</table>

Table 2. Run-time in milliseconds ($m = 3, r = mv$)
Table 3. Run-time in milliseconds ($m = 5, r = mv$)

<table>
<thead>
<tr>
<th>$m$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>16</td>
<td>80</td>
<td>2325</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>24</td>
<td>292</td>
<td>3958</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>4</td>
<td>8</td>
<td>51</td>
<td>723</td>
<td>8590</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>4</td>
<td>9</td>
<td>67</td>
<td>1082</td>
<td>13317</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>4</td>
<td>16</td>
<td>110</td>
<td>1621</td>
<td>19391</td>
</tr>
</tbody>
</table>

Table 4. Run-time in milliseconds ($m = 10, r = mv$)

4 Conclusions

The present research was conducted as part of a semester-long project by the first author in her junior year, and there are several opportunities for follow-up work. To begin with, our experiments yield tolerable run-times only when $L$ is quite small ($L \leq 18$ for $m = 2, L \leq 12$ for $m = 3, L \leq 8$ for $m = 5, \text{and } L \leq 5$ for $m = 10$). This may be suitable for the kinds CFGs found in typical FLA courses, but we propose to implement the incremental SAT solver approach in [9] for the next version of our tool, which is vastly quicker. We noted in the introduction that our CFGs are internally converted into CNF using a more efficient process than the one used by JFLAP. However, as pointed out in [10], Binary Normal Form (2NF) is an even more compact normal form (the normalized CFG is at most a constant factor larger than the original CFG). We therefore propose to augment JFLAP with a 2NF normalizer for CFGs, and use this in our implementation. Lastly, we can enhance the functionality of Batch Test for Equivalence by recording outputs in a log-file, instead of displaying the result of each test sequentially.

5 Acknowledgements

This research work was funded by NIIT Ltd., and was conducted wholly at NIIT University, Neemrana.

6 References


