Programming Project: DTD Validator for XML

Issa Memari

Keywords: Formal Languages, Automata, Thompson's construction

1. Introduction

The aim of this project is to write a program that is able to check whether an XML document is well-formed and valid with respect to a given DTD. For the first task, the solution is simple: bracket matching. However, the second task is much more complex, as it requires constructing an automaton for each regular expression in the DTD file and using it to match lists of sibling elements in an XML document.

Another difficulty lies in the fact that XML documents can be very large, hence we shouldn't assume that they can fit in memory. We implement the algorithms in streaming mode, that is, instead of reading the whole XML document and storing it in an in-memory representation like a tree, the XML documents are scanned and processed line by line. The checks for both well-formedness and validity are done in one scan of the XML document.

2. Transforming Regular Expressions into Automata

One of the methods for implementing regular expression matching is constructing an automaton that recognizes the same language as the regular expression, then using the automaton as the matching mechanism. There are two commonly used methods for constructing automata from regular expressions: Glushkov's construction and Thompson's construction. The automaton obtained by Glushkov's construction is the same as the one obtained by Thompson's construction algorithm, once its -transitions are removed. Therefore, since Thompson's construction is easier to implement, we implement Thompson's construction.

The main idea behind Thompson's construction is simple: it is possible to apply operators on automata. For example, to obtain an automaton that recognizes the language defined by the regular expression $(bc)^*d$, first build three automatons a_1, a_2, a_3 that recognize the languages $L_1 = \{"b"\}, L_2 = \{"c"\}, L_3 = \{"d"\}$ respectively. After that, apply the concatenation operator to automatons a_1, a_2 and label the resulting automaton a_4, then apply the Kleene star operator on a_4 and concatenate it with a_3.

In Thompson's construction, each automaton is assumed (without loss of generality) to have only one start state and final state. The operators are defined as follows:

- Concatenation of two automatons a_1 and a_2: add an ϵ-transition from the final state of a_1 to the start state of a_2, then change the final state of a_1 to not final.

- Kleene star of an automaton a_1: add an ϵ-transition from the start state to the final state of a_1, and another from the final state to the start state of a_1.
- Plus operator for an automaton a_1: add an ϵ-transition from the final state to the start state of a_1.
- Question mark operator for an automaton a_1: add an ϵ-transition from the start state to the final state of a_1.

The process of constructing an automaton using Thompson's construction is very similar to arithmetic expression evaluation; the numbers in arithmetic expressions correspond to minimal automatons that accept only one character in regular expressions and the arithmetic operators in arithmetic expressions correspond to Thompson's operations in regular expressions. Therefore, it would be convenient to convert the regular expression from infix to post-fix notation to make evaluation easier. For example, the regular expression $(ab)^+c*de?$ when converted to post-fix notation becomes $ab\cdot +c*de?\cdot\cdot\cdot$ where $\cdot$ is the concatenation operator. The algorithm for Thompshon's construction using post-fix regular expressions is described in Algorithm 1.

Matching strings to regular expressions using DFAs is much more efficient than matching using NFAs. Therefore, since automata that result from Thompson's construction are non-deterministic, we convert them to deterministic finite automata using the power set construction.

3. Validation

Once we have an automaton for each rule in the DTD file, we can validate an XML document against it. In a non-streaming setting, it is straightforward to do that; read the whole XML document into an in-memory tree, then traverse the tree and for each element assert that the string formed by the list of its children matches the corresponding regular expression in the DTD. However, in a streaming setting where we have to read the document line by line, it is a bit more difficult.

In order to be able to validate an XML document while reading it line by line, we create pointers to the current state in the DFA for each level of the tree as we scan through the file, and the pointers are pushed into a stack. This works as follows: when we encounter an opening element, we advance the top state in the stack according to this new element, then push into the stack the start state of the automaton corresponding to the new element, and when we encounter a closing element, we pop a state from the stack and assert that it is final. If at any point during the execution of this process it fails, then we report that the document is invalid.

4. Benchmarking

In order to test the implemented algorithms, we generated two sets of documents, the first is 500 large random XML documents that conform to a specific DTD, totalling 62

Algorithm 1 Thompson's Construction

```
1: procedure ConstructNFA(regex)
2:     $stack \leftarrow \phi$
3:     for $i < regex.length$ do
4:         if $regex[i] = \cdot$ then
5:             $a_1 \leftarrow stack.pop()$
6:             $a_2 \leftarrow stack.pop()$
7:             $a_3 \leftarrow a_2 \cdot a_1$
8:             $stack.push(a_3)$
9:         else if $regex[i] = +$ then
10:            $a_1 \leftarrow stack.pop()$
11:            $a_1 \leftarrow a_1^+$
12:            $stack.push(a_1)$
13:        else if $regex[i] = *$ then
14:            $a_1 \leftarrow stack.pop()$
15:            $a_1 \leftarrow a_1^*$
16:            $stack.push(a_1)$
17:        else if $regex[i] =?$ then
18:            $a_1 \leftarrow stack.pop()$
19:            $a_1 \leftarrow a_1^?$
20:            $stack.push(a_1)$
21:        else
22:            construct $a_1$ that accepts the language $L = \{regex[i]\}$
23:            $stack.push(a_1)$
24:    return $stack.pop()$
```

GB in size. The problem of generating random documents that conform to a given DTD boils down to generating strings that conform to a given regular expression. Fortunately, an open-source java library called Xeger does that for us.

The algorithm for generating random XML documents that conform to a given DTD in a streaming setting is somewhat similar to the algorithm for validation. The idea is to write an opening element (the root) to the stream, then generate a string that matches the regular expression corresponding to the element, and for each character in that string repeat recursively. When the end of a string is reached, write a closing element to the stream.

Figures 1 and 2 show the plots for execution time and memory usage with respect to document size. Execution time is clearly linear with respect to document size. On the other hand, there is no clear correlation between memory consumption and document size, this is because the algorithm is operating in streaming mode, that is, the XML documents aren't stored in memory.

However, there should be some correlation between memory consumption and document depth, because we store a pointer to the current state in the DFA for each level of the tree. To test that, we generated another set of documents containing 500 left linear XML documents of different depths that conform to a simple DTD. Figures 3 and 4 show the plots for execution time and memory usage with respect to document size.

We notice that in figures 1 and 4, the samples form multiple lines. This is probably due to the JVM doing some runtime optimization.

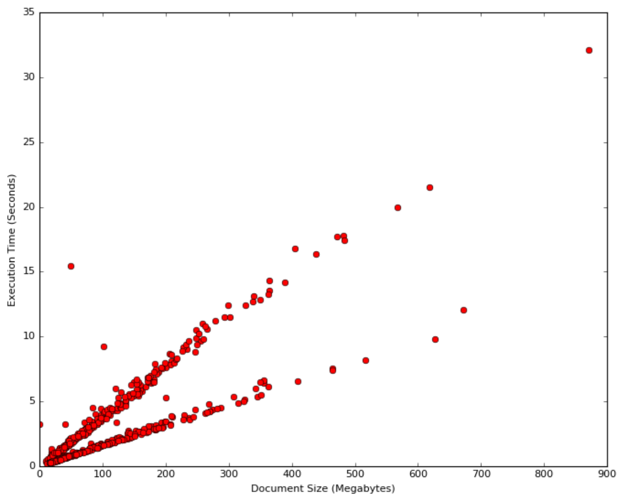

Figure 1: Execution time: random documents

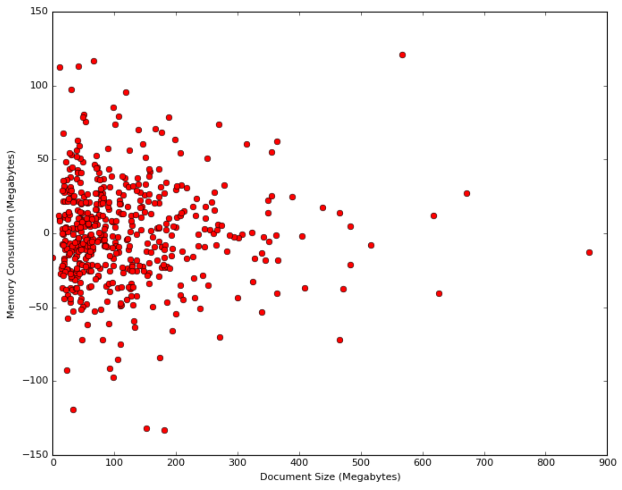

Figure 2: Memory usage: random documents

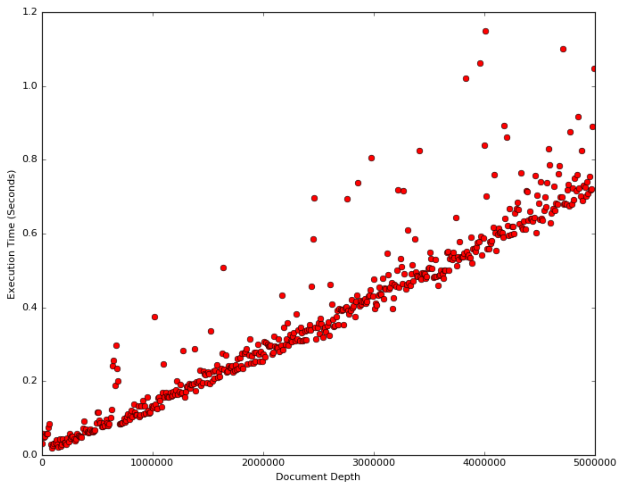

Figure 3: Execution time: left linear documents

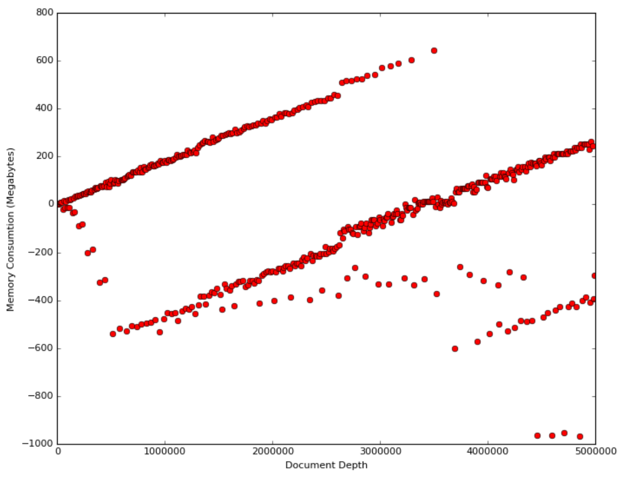

Figure 4: Memory usage: left linear documents

5. Conclusion

The implemented method to validate XML documents against a given DTD seems to have very good performance both in terms of execution time and memory usage. However, we did not implement validation using an in-memory representation, therefore we are not able to compare execution time for the two methods. We hypothesize that execution time will be better if using an in-memory representation because because in streaming there is IO overhead, and this remains to be experimentally verified.

6. Contents of Submitted Folder

Java was used to implement the project. The compressed file "DTD Validator for XML.tar.gz" contains the netbeans project, the source code, the executable jar file, the DTD files used to generate the test sets and the results of the experiments as CSV files. To run the program in the command line, execute the following:

```
java -jar DTD_Validator_for_XML.jar xml_file_path dtd_file_path
```