WEB DATA MODELS

PROGRAMMING PROJECT

DOCUMENTATION

Professor:
Silviu Maniu

Student:
Nedeljko Radulovic

Contents

1. Problem explanation

The goal of this work is to implement checking if the given XML is well-formed and then to validate that XML on a given DTD file. Checking if the XML is well-formed consists checking if xml file has only one root, if all elements are properly nested and if the XML satisfies given form:

0|1<whitespace> element1
0|1<whitespace> element2
0|1<whitespace> element3

Where 0 – means opening new element, 1 – closing the element and element1, element2… present the name of the element. Simplified example of the XML file:

0 a
0 b
0 c
1 c
1 b
1 a

Second part of this project is to implement matching function for regex in order to check if the XML is valid over given DTD file. DTD file is also given in simplified form:

element1<whitespace>regex1
element2<whitespace>regex2
element3<whitespace>regex3

For example:

a b
b c*
c _

Also, regex can be empty and then it is represented with underscore: '_'

2. Implementation

Implementation has been done in Python programming language. Program first checks if XML file is well formed and if it is well formed it builds a corresponding tree structure for that XML file. If XML is well-formed program checks for its validity. For that purpose, the Deterministic finite automaton is constructed for every regex in DTD file and then the every element in XML tree is validated on its corresponding automaton.

The program takes two arguments: *file.xml* and *file.dtd.* And after running the program prints on standard output two lines:

1. If XML is well-formed: “XML is well-formed.” otherwise “XML is not well-formed.”
2. If XML is valid “XML is valid.” otherwise “XML is not valid.”

2.1. XML well-formed?

Checking if XML is well-formed consists of checking if every line in XML file is in the form stated above.
The program reads the XML file and first, to shorten the time of execution in case that root-element is not configured well, checks if the first and last line in XML file represent opening and closing of the same element, respectively. This is all done in method called *XmlWellForm(“file.xml”).*
After this first check is done, and if it is satisfied, program proceeds with checking the whole XML file and building a corresponding tree on the fly.
Building a tree is done using a stack and a specially defined classes for tree and for tree node:

```
class Tree(object):  # Class Tree - Nodes belong to a tree
    def __init__(self):
        self.rootNode = None  # Define Root Node
        self.nodes = []  # List of nodes that are in the tree

    def show(self):
        for node in self.nodes:
            print(node.data, node.children)
            # Print nodes and their children

    def addNode(self, object):
        self.nodes.append(object)
        # Add new node to a tree
```

Picture 1. Class Tree

Object of class Tree consists of RootNode – which is object of class TreeNode and represents root node of the tree, and nodes - which represents a list nodes in the tree, which is a list of objects of class TreeNode.

```
class TreeNode(object):
    # Define class TreeNode - We will be making a tree, and therefore we define this
class for nodes of the tree

    def __init__(self):
        self.data = None
        self.parent = None
        self.children = []
        # Every node has data - name of the element (a, b, c...)
        # Parent: represents parent node
        # Children: List of elements (a, b, c ...), which are children of current node

    def addChild(self, object):
        self.children.append(object)
        # Add new child: Append name of the element to the list

    def addParent(self, object):
        self.parent = object
        #Add parent: Node that is parent to current node
```

Picture 2. Class TreeNode

Object of class TreeNode has following properties:

- data – name of the element in the XML
- parent – object of class TreeNode which represents parent node to a current one
- children – a list of child nodes e.g. list of objects of class TreeNode

As mentioned before, tree is built using a stack, so we defined new class Stack:

```
class Stack:  # Define new class - Stack
    def __init__(self):
        self.stacklist = []

    def size(self):
        return len(self.stacklist)
        # Returns size of the stack e.g. length of a list

    def isEmpty(self):
        return self.size() == 0
        # Check if stack is empty, returns True if stack is empty

    def push(self, newItem):
        self.stacklist.append(newItem)
        # Add element on top of stack

    def pop(self):
        self.stacklist.pop()
        # Take out the top element from stack

    def peek(self):
        return self.stacklist[self.size() -1]
        # Peek, check the element on top of stack but do not remove it from stack
```

Picture 3. Class Stack

The algorithm of making a tree is following: For every line in XML file we read name of the element and if the element is opening or closing. If it is opening element we stack the corresponding closing element.
For example: If we read: **[0 a]** we stack **[1 a].** As it will be easier later to check if we are closing the proper element. As we opened new element, we create new node in a tree, assign corresponding properties and update parent and children properties. If we read closing element, we check if it matches the top of the stack and if this is true we pop out of the stack. The algorithm finishes when we have both: empty stack and end of XML file. If we have only one of these then XML is not well formed. This algorithm can be represented with following pseudocode.

1. *xml_tree =* ***create****(NewTree)*
2. ***For every line*** *in XML [state, element_name] = read(line(0), line(1))*
3. ***If*** *state == "0" – opening element*
4. ***stack.push****(["1", element_name])*
5. *new_tree_ode =* ***createnew****(TreeNode)*
6. ***if*** *firstNode: rootNode = newTreeNode, firstNode ->* ***False*** *-- firstNode – Boolean flag*
7. ***Update*** *children and parent properties*
8. ***Add*** *new_tree_node to xml_tree*

```
9.  Else if state == "1"  -- closing element
10.     if [state, element_name] == stack.peek()
11.             stack.pop()
```

This algorithm is performed in method ***XmlWellForm,*** and it returns an object of class TreeNode, representing the resulting tree built based on a given XML file. As we perform checks during the building of a tree, if the XML is not well-formed this method returns ***None.*** After calling this method we simply check if the result of the method is not ***None*** in order to confirm if XML is well-formed.

2.2 XML valid?

Well-formedness of XML is in fact weak requirement and it says only that that XML can be represented in form of a tree. Checking if XML valid means that it must conform a special dialect, in our case it is DTD. DTD is a file where it is specified how every element in XML can look like. In a simple words, and it is going to be used later in algorithm, DTD describes for every element in a tree what is the allowed sequence of child elements. So, for example, if our DTD says: **a (bc)*,** that means in these simple words: element **a** can have as a children zero or more sequences of element **b** followed by element **c**. And this is specified using regular expressions. In order to be able to check the XML for validity over given DTD we have to transform regular expression into Finite State Automaton.

For this there is several solutions. One solution is to build a Glushkov automata for the regex, and the one chosen in this work is Thompson's construction. More words on algorithm will be later.

2.2.1. Postfix notation

In order to follow the algorithm to construct the Thompson construction first, we need to convert regular expression from infix to postfix notation.

$$Infix: a.(b.c)* \rightarrow Postfix: abc.*.$$

Where '.' represents concatenation operation. Converting regular expression from infix to postfix notation also takes care of grouping operator, so in postfix notation we don't have explicitly grouping operator.

In this program we use two separate methods: ***InsertDot(regex)*** and ***regex2post(regex_with_dot)*** to convert regular expression to postfix notation. In ***InsertDot*** method we just put dots '.' in place where should be concatenation operator. And in ***regex2post*** method we convert regular expression from infix notation to regular expression in postfix notation. For this we also use stack. This algorithm can be expressed through pseudocode:

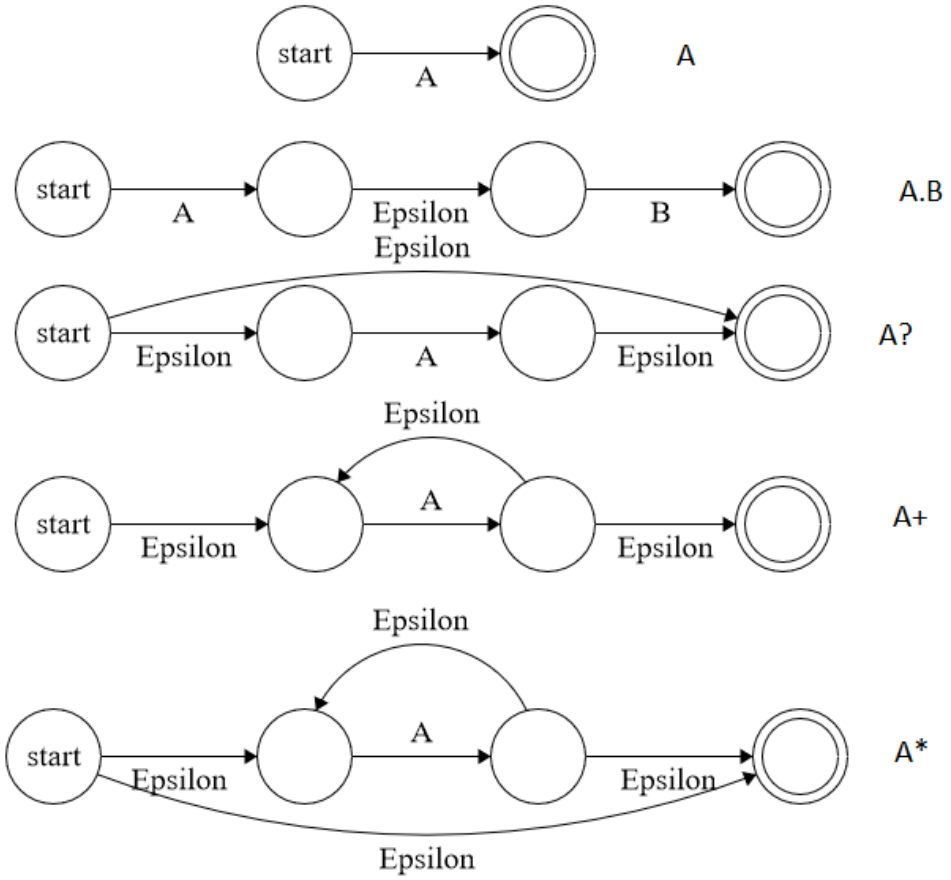
```
1.  stack = [], infix_regex = input_string, postfix_regex = ‘’
2.  for each char in infix_regex:
3.      if char is literal: postfix_regex <- char
4.      if char is operator:
5.          if stack is empty: push(char)
6.          else operator is not grouping:
7.              Compare precedence of operator
8.              The one with higher precedence -> postfix_regex
9.              The one with smaller precedence -> push()
10.         else if operator is “)”:
11.             while not (pop.stack == “(”)
12.                 postfix_regex <- pop.stack()
13. while stack is not Empty: postfix_regex <- char
```

Now, after converting regular expression from infix to postfix notation, next step is to create NFA.

2.2.2. Building NFA

As we already mentioned we will be building NFA using Thompson’s construction.

Thompson’s construction is the algorithm of transforming regular expression into equivalent Non-Deterministic Finite Automaton (NFA). The idea of the Thompson’s construction is to have a set of rules to represent literals and operators in regular expression. For every literal and operator a fragment is being constructed in that way that we always have one start state and one final state.



Picture 4. Set of rules

In picture 4. are displayed rules for converting regular expression to NFA. In our work we assume that operators used in regular expressions are: '.', '?', '*', '+', '( )'.

In our program, creating NFA is done in method ***post2NFA(postfix_regex).*** This method takes as argument regex in postfix notation and builds NFA based on rules on picture 4. When creating new state in NFA, we define the transitions from this state. Algorithm says, that we always have one start state and one final state, so for every operation we add some new states. Adding these states will impact on new empty, so called *"Epsilon"* transitions. These transitions represent possible ways to go from one state with empty transitions.

For this algorithm we define two new classes:

```
class State(object):
    def __init__(self, name = None):
        self.name = name
        self.transitions = defaultdict()  # Following state
        self.transitions[eps] = set()


class Fragment(object):
    def __init__(self, startState, finalState):
```

```
        self.startState = startState
        self.finalState = finalState
        self.literals = []
        self.states = []
```

Picture 5. NFA classes: State and Fragment

Object of class state represents a state(node) in NFA. Important property of this node are transitions. Transitions are represented with *dictionary* and the key is the transition letter or ‘eps’ for empty transitions. As every state can have several empty transitions, we keep them all under the same key and as a value we put set of states to which those transitions point.

And object of class Fragment represents NFA during his building and also when it is done. For NFA we save start state and final state as this is very important because, as already mentioned, while building the NFA we always have only one start state and one final state. Also, we keep list of literals, that means letters that appear in the regular expression and list of all states in NFA. These properties may be handy when we are constructing DFA out of NFA.

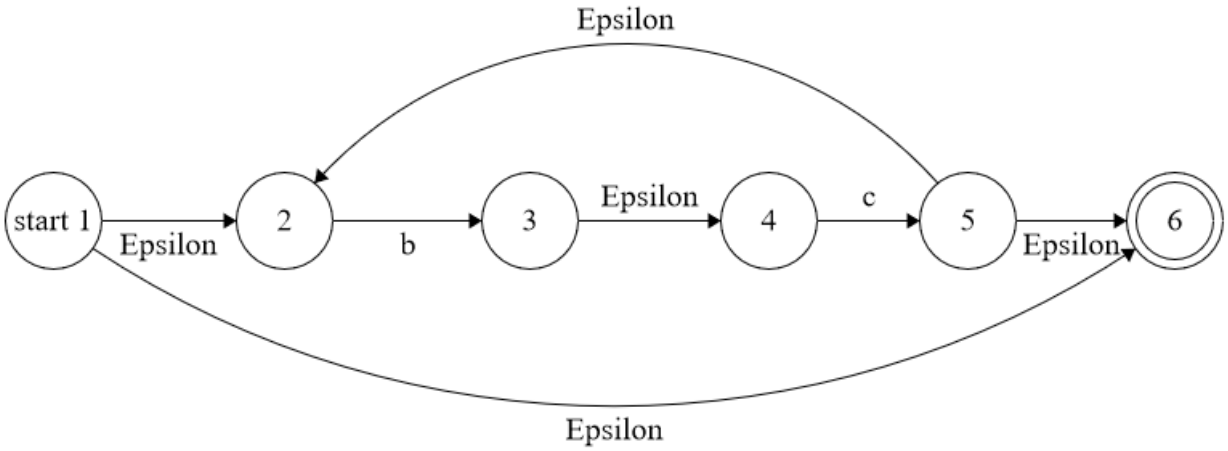
Complexity of building NFA is linear in time O(m), where m is number of nodes, but matching the string on NFA is of complexity O(mn) where n is the length of string. But this way is costly in that way that in NFA more than one node can be active at each step and all need to be updated. Better solution is to convert NFA do Deterministic Finite Automaton(DFA). Creating the DFA out of NFA will have complexity O(2^m) but then matching string of length n on DFA will have complexity of O(n).

2.2.3. From NFA to DFA

Once when we have built the NFA, in order to simplify matching of string, as mentioned before, we will construct DFA. DFA has the property that from one state we have exactly one transition on one literal and we don’t have empty transitions. First part of this algorithm is to calculate e-closure sets for all states in NFA. E-closure is the set of states to which there is empty – “Epsilon” transition. For every node there is always empty transition to him self. Lets explain the algorithm on a example:

Lets take following regular expression in postfix notation: $bc.*¿$

Applying rules from picture 4. NFA for this regex looks like this:



Picture 6. NFA for bc.*

E-closure for one node represents set of nodes to which you can go on epsilon transitions, one or multiple, so in our example:

$$E-closure(1)=\{1,2,6\}$$

As we have epsilon transitions from node 1 to nodes 2 and 6, but as mentioned before, every node has an empty transition to himself. So E-closure for the rest of the nodes in our example are:

$$E-closure(2)=\{2\}, E-closure(3)=\{3,4\}, E-closure(4)=\{4\},$$

$$E-closure(5)=\{5,2,6\}, E-closure(6)=\{6\}$$

In our program this is done with recursive method ***e_closure(state, visited)***

```
def e_closure(state, visited):
    global eps
    if (state not in visited) and state.transitions[eps]:
        #print(visited)
        for item in state.transitions[eps]:
            visited.add(item)
            e_closure(item, visited)
    else:
        visited.union(state.transitions[eps])
    return visited
```

Picture 7. e_closure method

In this method for each node in NFA we follow epsilon transitions from this node and store visited nodes. We keep track of nodes visited so we don't do the same nodes twice. **visited** represents e-closure set for one node.

Next step in building DFA is to use transition on letters for each node and e_closure and calculate new transition table. Now we define new start state, start state for our DFA. In our case it will be {1,2,6}.

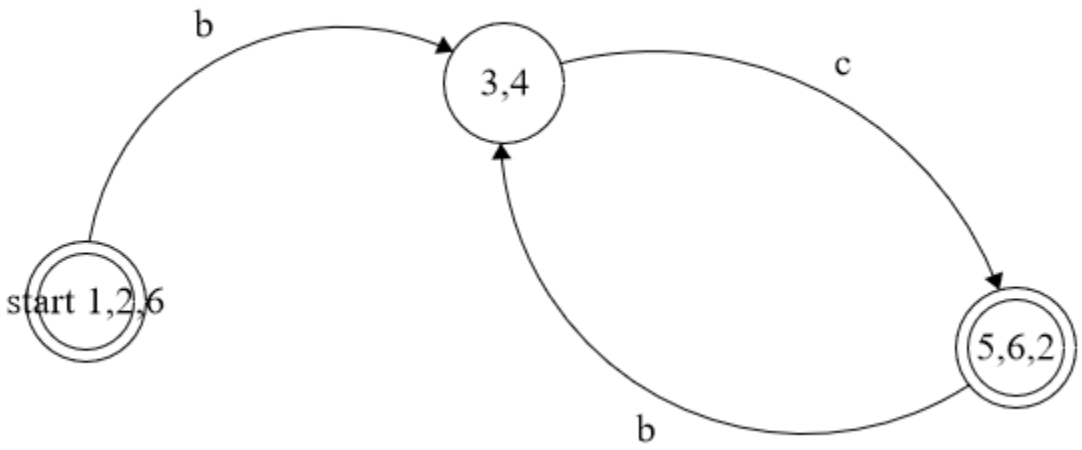
We need to find all possible transitions from these states on every possible letter. We will show this part of the algorithm through table:

	b	c
{1,2,6}	{3,4}	∅
{3,4}	∅	{5,6,2}
{5,6,2}	{3,4}	∅

Table 1. DFA states

So, as we have start state {1,2,6}, for every state in this set we check if we have transitions for letters. In our case, we have to letters b and c. We first check for b transitions in states 1, 2, 6. For 1 and 6 we don't have b transitions but we have b transition for state 2 (Picture 6.). On b transition we can go from state 2 to state 3, but also, from state 3 we can go to state 4 on empty transition that is why we add also 4. We check the same for letter c, but in this case we don't have possible transitions. After this step we check, and in the program we do it with a stack, if {3,4} is already added as a new state. If not, we add it and on this table it means that we add it on the left side. Now we repeat the same process for this new state.

We stop when we don't have anything to add on the left side more. Now we need to identify starting state and accepting state for our DFA. We already said that the start state is {1,2,6} and accepting states are all the states that contain final state from NFA. In our case final state from NFA is 6, so accepting states for DFA are: {1,2,6} and {5,6,2}. And our DFA looks like this:



Picture 8. DFA

In program we use method ***makeDFA*** to implement this algorithm and build DFA. We define new classes for DFA and DFA state:

```
class DFA(object):
    def __init__(self):
        self.startState = DFAState
        self.dfa_states = set()


class DFAState(object):
    def __init__(self):
        self.name = None
        self.transitions = {}
        self.accepting = False
        self.list_of_nfa_states = []

    def __eq__(self, other):
        return self.list_of_nfa_states == other.list_of_nfa_states
```

Picture 9. DFA Classes

For every object DFA we have his start state and set of states. And for every DFA state we have set of transitions with key as literal and value as a DFA state to which it points. And we have list of NFA states that are part of this DFA state. Also for every state we have a property saying if the node is accepting or not.

2.2.4. Matching a string

As a last part we need to use all of these implemented algorithms to match a string on a regex. For this we define method ***match(children, dfa_start_state).***

Arguments for this function are list of children for a element from XML file which is referenced in dtd file and dfa_start_state is the starting state of DFA made on a regex for that element.

List of children is list of chars that represent element names. So we first convert this to a string and then do the run on DFA. If we reach the end of string and we are in accepting state we have a match and XML is valid, if not XML is not valid.

3. Experimental results

We have already discussed the complexity of the algorithms used in this program but we also need to do some real test in order to prove that the execution time is exponential and also to check memory consumption.

3.1. Experimental settings

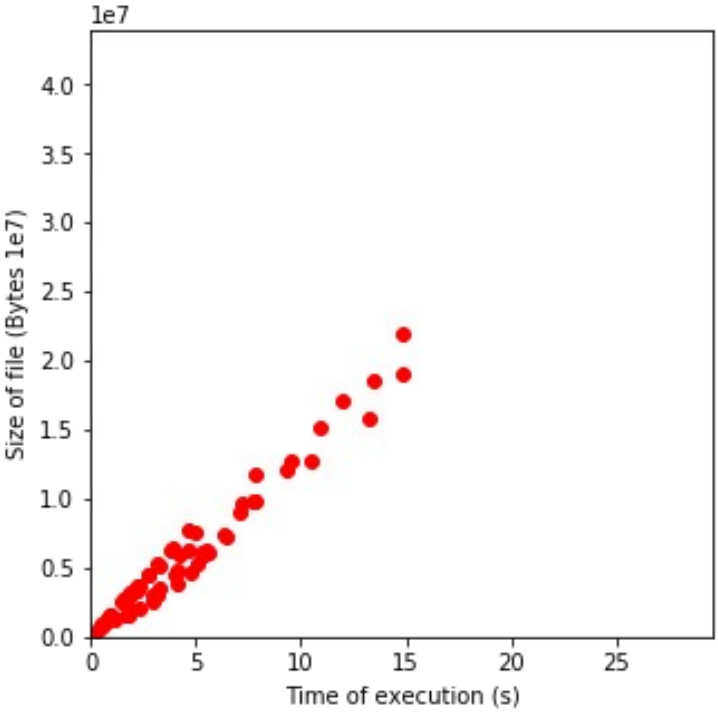
For testing purposes we have generated xml files and dtd files with random number of states, depth and length of regular expressions. We run the test and use following libraries to store execution time and memory consumption:

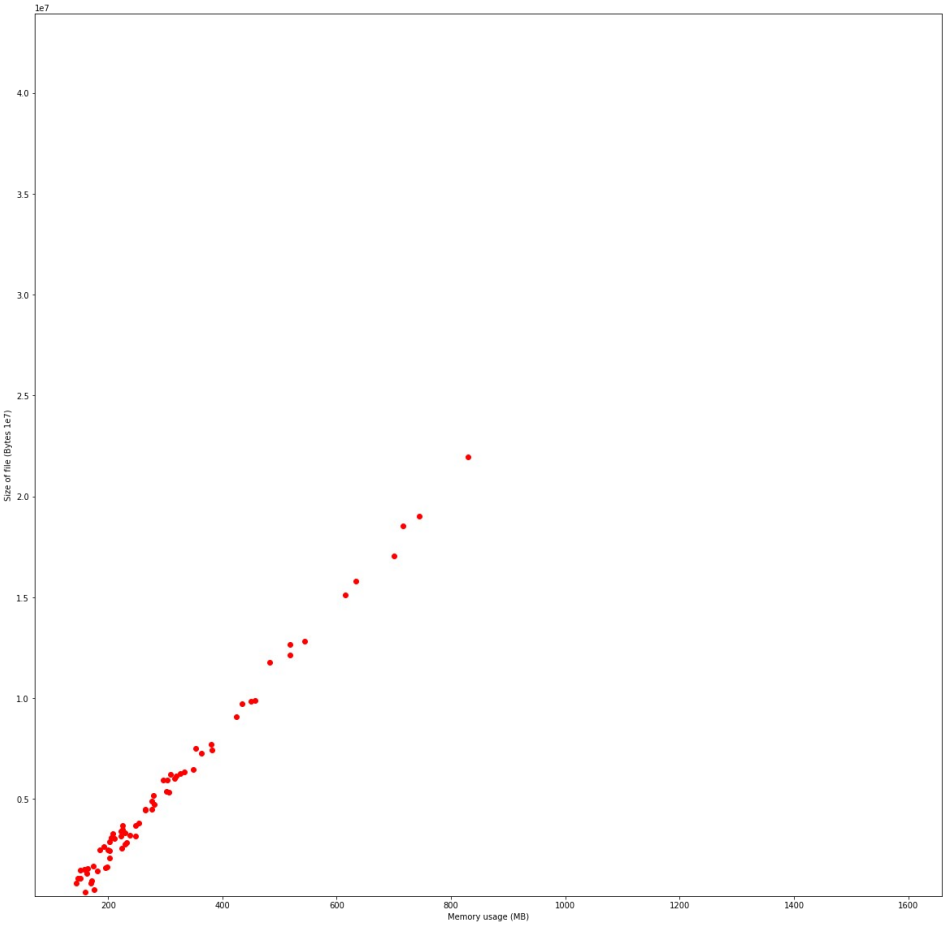
1. timeit.py - for execution time
2. memory_profiler – for memory consumption
3. matplotlib – for plotting

Tests are run on: **Ubuntu 16.04 LTS, 12GB RAM, Intel® Core™ i7-4720HQ CPU @ 2.60GHz**

3.2. Results

For experimental result we have generated random xml files and ran the program on them. Below are shown the results:


1e7
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
Size of file (Bytes 1e7)
0
5
10
15
20
25
Time of execution (s)


1e7
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
Size of file (Bytes 1e7)
200
400
600
800
1000
1200
1400
1600
Memory usage (MB)