WEB DATA MODELS

PROGRAMMING PROJECT

DOCUMENTATION

Professor: Silviu Maniu **Student: Rakhmetzhanov Meirkhan**

Contents

1. Problem explanation

The objective of this programming assignment is to be able to implement and analyze efficient algorithms for fragments of XPath queries.

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

where 0|1 is a bit indicating a startElement or endElement event respectively, and element is the name of the element. For instance, the following file example.txt contains the document <a>:

0a
0b
1b
1a

The parameters of the implemented executables will be the following:

./<program_name > <xml_file > <xpath_query >

The console output will be the preorder ID (zero-indexed) of the matching nodes, one per line, or will be empty if no nodes match. For instance:

./<program_name > example.txt //a/b

will return the following output:

1

The goal of this work is to Implement a streaming algorithm for XPath queries of the form

//e1/e2/ · · · /en,

1

where **ei** are element names.

Second part of this project is to Implement a streaming algorithm using the lazy DFA method, for queries of the form:

//p1//p2// · · · //pn,

where each **pi** is an path of the form:

ei1/ei2/ · · · /eim,

and **eij** are element names.

2. Implementation

Implementing XML Stream format means processing document line by line.
Implementation has been done in Python programming language.
Program takes two arguments: *file.txt* and xpath query and prints matched nodes of xml tree line by line. Given xpath query defines further direction of program: *simple* or *complex*

```
def preprocess():
    """
    preprocessing of input parameters
    :return:
     algortihm - string represesnting complexity of algorithm
     query_list - query given by user, splitted to list
    """
    # input_file = '/Users/meirkhan/Downloads/m2_dk_wdm_project_example/input.txt'
    # query_xpath = '//a//b//a//b//c'

    input_file = sys.argv[1]
    query_xpath = sys.argv[2]

    query_complexity = query_xpath.count('//')
    if query_complexity > 1:
        algorithm = 'complex'
        query_xpath = query_xpath.replace('/', '')
        query_list = list(query_xpath)
    else:
        algorithm = 'simple'
        query_list = query_xpath[2:].split('/')
    return algorithm, query_list, input_file
```

Picture 1. Defining complexity of query

Main method of program responsible for preprocessing input parameters (Picture 1) and further line by line processing.

```
def main():
    try:
        algorithm, query_list, input_file = preprocess()
        streaming_instance = StreamParser(algorithm, query_list)

        with open(input_file, 'r') as file:
            for whole_line in file:
                line = whole_line.split()
                streaming_instance.process_line(line)

        # print matched nodes line by line
        for node in streaming_instance.result:
            print(node)
    except:
        print('Please, check your input file or query for correctness')
```

Picture 2. Main entry point of program

All main functional variables and methods are belong to instance of class StreamParser

```
class StreamParser():
    """
    Class which processes simple xpath query
    """

    def __init__(self, algorithm, query_list):
        self.stack = Stack()
        self.result = []
        self.node_number = 0
        self.algorithm = algorithm
        self.query_list = query_list
```

Picture 3. StreamParser class

2.1. Implementation of Simple Xpath query

In this work as "simple query" we are considering linear xpath queries each state of referring to direct child (*//e1/e2/ · · · /en*).

Processing of both "simple" and "complex" algorithms implemented within stack. Stack structure necessary to avoid keeping whole tree in memory. Therefore, while processing any node, stack will keep only ancestors (items with *startElement ID*) of current node. Already processed nodes will be deleted from stack with reaching *endElement ID*.

```
class Stack:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def peek(self):
        return self.items[len(self.items) - 1]

    def size(self):
        return len(self.items)

    def get_element(self, index):
        return self.items[index]
```

Picture 4. Implementation of stack

For every node, as soon as program gets opening tag(*startElement ID = 0*) there is test for matching current node with given Xpath query.

Every matched node number will be stored in resulting array.

To avoid excessive checking only nodes located deep enough in tree to match query structure will be checked.

Node matches with query if (Picture 5):

- all of **last n elements** of stack (including current node) matches with corresponding query states
- **n** - number of elements in query

```
def check_match_simple(self):
    """
    Method to check whether current node matches given query (for simple queries)
    :return: True if node matches to given query, False otherwise
    """
    match = True
    for i in range(len(self.query_list)):
        if (self.query_list[len(self.query_list) - 1 - i] !=
                self.stack.get_element(self.stack.size() - 1 - i)):
            match = False
            break
    return match
```

Picture 5. Function for checking matches for simple query

2.1. Implementation of Complex Xpath query

In this work as “complex query” we are considering linear xpath queries each state of referring to all descendents.

Implementation and used data structures for “complex” queries same as for “simple” queries. Except test for matching.

As soon as program gets input Xpath query, it turns query into array of elements, ignoring *axes. E.g* query **//a/b//a** becomes **[a,b,a].**

For every node, as soon as program gets opening tag(*startElement ID = 0*) there is test for matching current node with given Xpath query.
Every matched node number will be stored in resulting array.
To avoid excessive checking only nodes located deep enough in tree to match query structure will be checked.

Node matches with query if (Picture 6):

- last **n “compressed”** ancestors of current node(including current node) matches with query array in corresponding order.
- **“compressed” nodes** - parent-child nodes having same element name and located next to each other in branch.

For instance,
query = **//a/b//a**
query list respectively = **[a,b,a]**
stack in current node = **[a,b,a,a,a]**

As soon as last 3 elements of branch (of stack accordingly) have same element name and located next to each other, we can **compress** them to single *a* element. Therefore, stack becomes **[a,b,a]** - same as query and this node is matched.

Program will search for next element in stack *while* it finds node different from current.

```python
def check_match_complex(self):
    """
     Method to check whether current node matches given query (for complex queries)
    :return: True if node matches to given query, False otherwise
    """
    match = True
    old_node = ''

    for i in range(len(self.query_list)):
        current_node = self.stack.get_element(self.stack.size() - 1 - i)
        if current_node == old_node:
            j = i + 1
            while current_node == old_node:
                current_node = self.stack.get_element(self.stack.size() - 1 - j)
                j += 1
                if j == self.stack.size():
                    break
        if (self.query_list[len(self.query_list) - 1 - i] != current_node):
            match = False
            break

        old_node = current_node
    return match
```

Picture 6. Function for checking matches for complex query

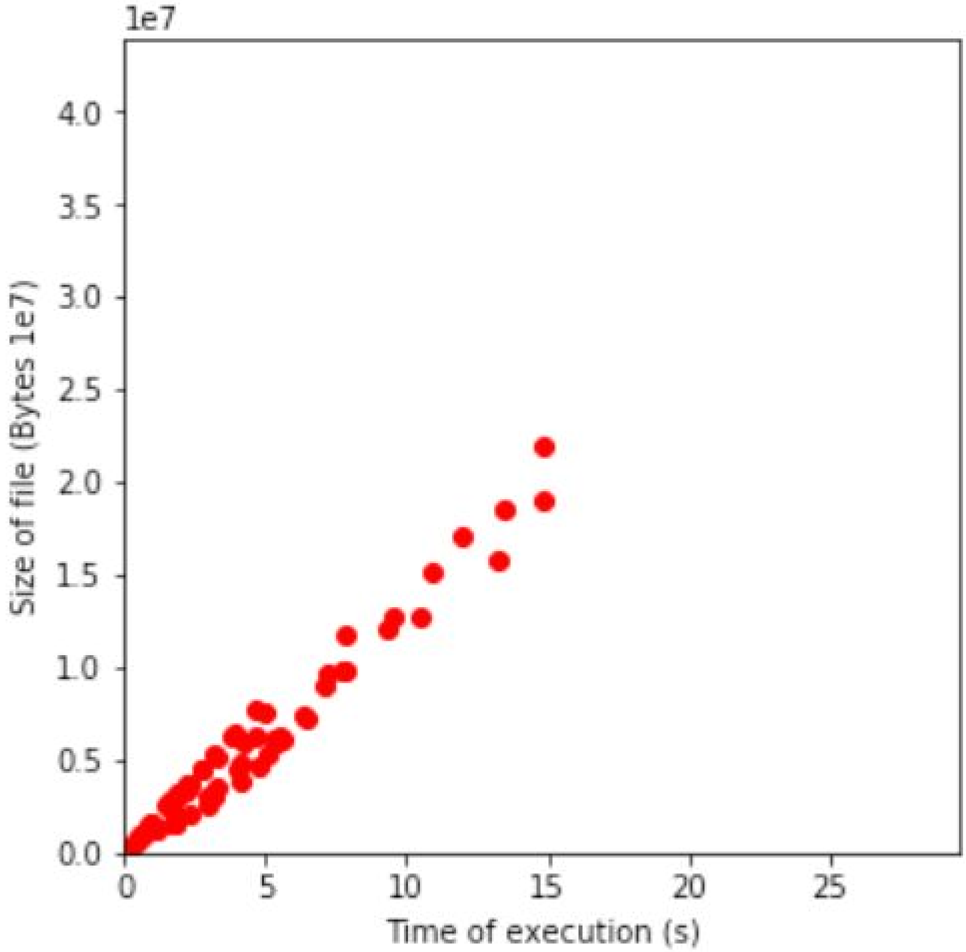
3.Experimental results

3.1. Experimental settings

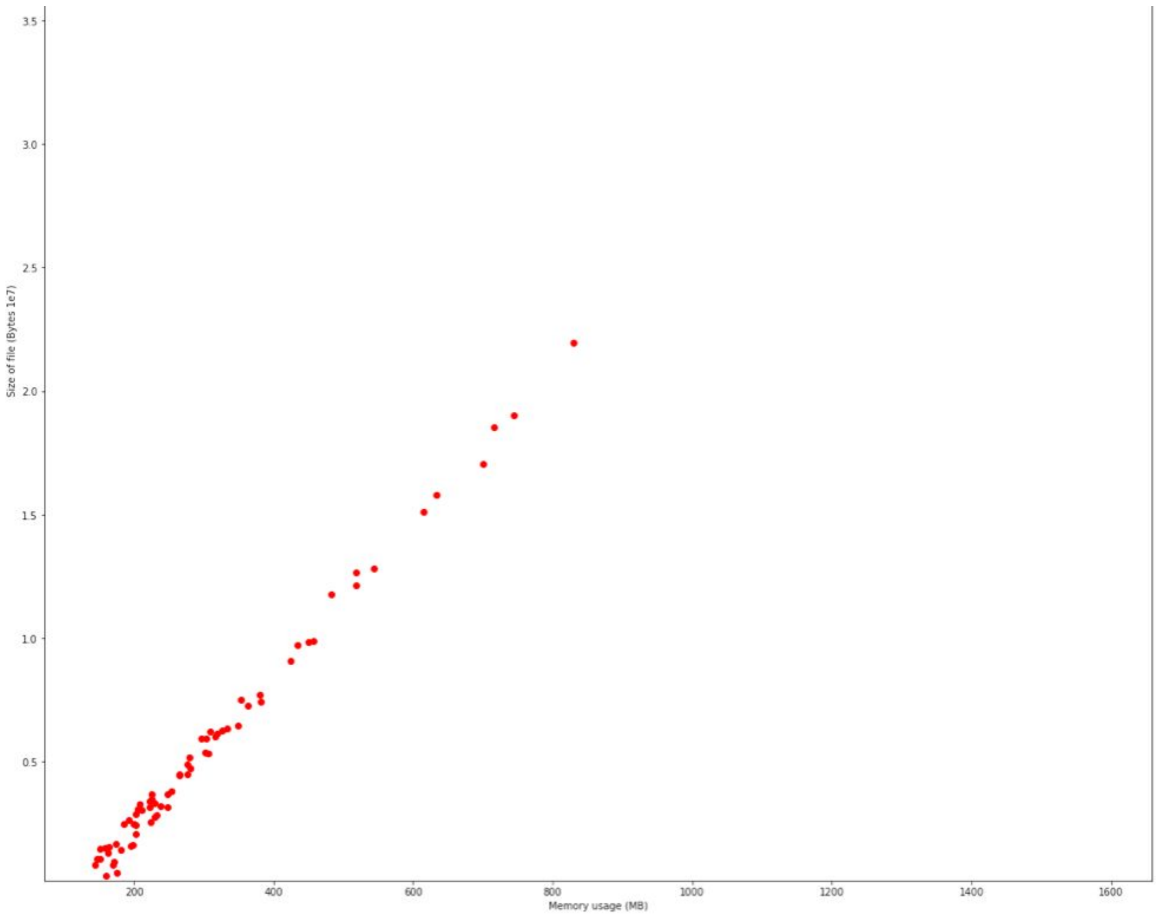
I run the test and use following libraries to store execution time and memory consumption:
1. timeit.py - for execution time
2. memory_profiler – for memory consumption
3. matplotlib – for plotting
Tests are run on: **Mac OS Mojave 10.14.1, 4 GB RAM, 1.3 GHz Intel Core i5**

Size of documents = number of nodes in XML tree.

3.2. Results



Picture 4. Size of document to time of execution



Picture 5. Size of document to memory size