Programming Project Steaming Algorithms for XPath

Web Data Model

ZHOU Peikun 2016/11/20

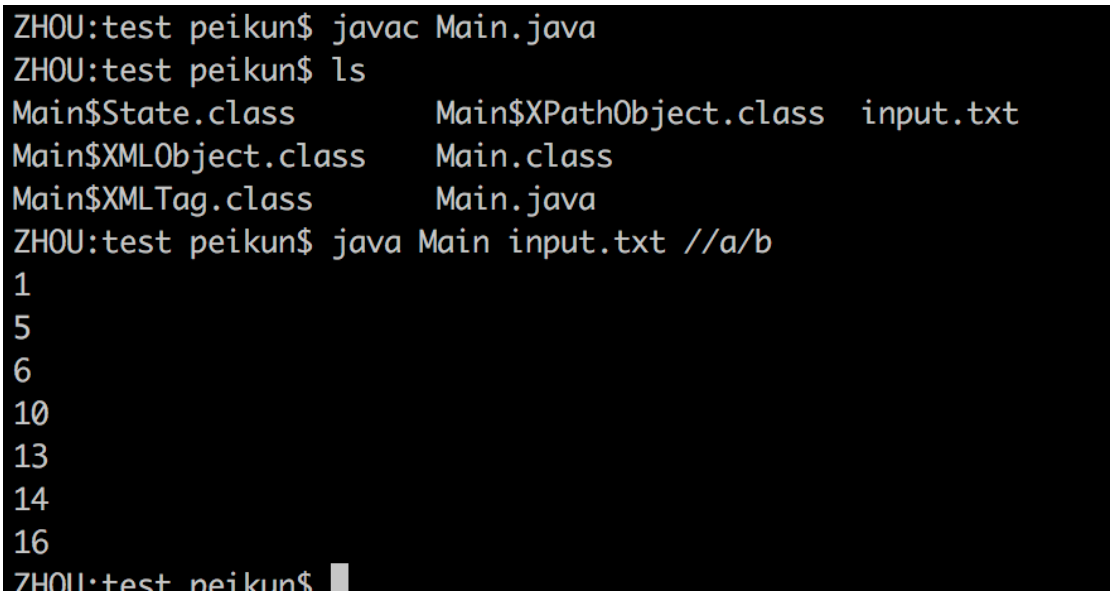
Supervisor: Silviu Maniu

1. How to execute

   The project is made up of java code, compiled class, and input text, so we can execute it without compile. Open shell at this folder, execute

   java Main input.txt <xpath-query>

   The example of executing *java Main input.txt //a/b* likes blow,



2. Algorithm Implementation

   a) Stream XML input

   For xml streaming input, I use java.io.* to read and get a StreamXML object. Then I iterate over each line and then split each line with " " to get XMLTag which includes nodes' name and weather it's opened(explored).

b) Query XPath

For the query XPath including "//", we can see it as //Q1//Q2//…, each query Q1 is a simple query path(/a/b/c..). So I remove the characters "//" and stored the query as String[][] paths; Then for each simple query of paths, I split "/"and get two dimension array Xpath.

c) Stamp & Stack

I create a class named Stamp(node, path) to store each step's node and path, just like the method told in class, each step of stamp is stored in a stack and the initial stamp is (0, 0), if a node isOpened/explored, we continue process the head element, or we remove it from head of stack until initial. Inside stamp, we judge weather Xpath is equal to node's name then analysis weather node and path meets the querypath. The Match Algorithm pseudocode is blow,

```
1  Algorithm match(XMLNode XMLnode, int node, int path)
2      int pathMatch, nodeMatch
3      if XPath[path][node] == tag.getName()
4      // query is completed
5          if XPath[path].length == node + 1
6              // each path is completed
7              if XPath.length == path + 1
8                  nodeMatch = queries[path][node]
9                  pathMatch = path
10                 System.out.println(order)
11             else
12                 nodeMatch = 0;
13                 pathMatch = path + 1;
14         else
15             nodeMatch = node +1;
16             pathMatch = path;
17         // push new stamp into stack
18         stack.push(new Stamp(nodeMatch,pathMatch))
19     else
20         nodeMatch = queries[path][node]
21         pathMatch = path
22         //
23         if(node !=0)
24             match(tag,nodeMatch,pathMatch)
25         else
26             stack.push(new Stamp(nodeMatch,pathMatch))
```

d) Algorithm KMP

Another important algorithm is KMP(the Knuth-Morris-Pratt algorithm), we usually use it to solve text pattern match problem. I learnt it with help of two sources, Pattern matching(cs.princeton.due), and Knuth-Morris-Pratt string matching(uci.edu). Be different with violent match, KMP is more efficient because it avoid invalid backtracking. Its pseudocode is blow,

```
Algorithm int KMP(char* s, char* p)
    int i = 0;
    int j = 0;
    int sLen = strlen(s);
    int pLen = strlen(p);
    while (i < sLen && j < pLen)
        // j=-1 or current string match, both i and j ++
        if (j == -1 || s[i] == p[j])
            i++;
            j++;
        else
            // j!=-1 and current string match failed, only change j
            // next[j] is j's next value
            j = next[j];
    if j == pLen
        return i - j;
    else
        return -1;
```

To use it in my code, I recode it with query array and path array. At first, we create an query array whose length is equal to path's length and its first value is -1. When path[count] == path[i-1], query[i]=count+1, and count++; else if count != 0 which means it has changed from initial, count = query[count]; else query[i] == 0. At last, we return query array. For each subpath query, execute KMP for e=them, then we get currently KMP query. Then we can do Stamp update with it, change the current path or don't change and continue explore.

3. Evaluation analysis

I evaluate its execute time for several queries with system.currentTimeMilles(), I found that the executed time is almost a constant, they're 2ms. The reason I guess may be my queries is not big enough.

For memory space, I use runtime.totalMemory()-runtime.freeMemory(), but I don't think it's precise enough, the result is around 783342k, 749845k, and 766761k.

4. References

a) http://www.cs.princeton.edu/~rs/AlgsDS07/21PatternMatching.pdf

b) http://www.ics.uci.edu/~eppstein/161/960227.html

c) https://github.com/cris-b/blacktie/blob/b11ea2ad464208624669121fad6a8bff5ecf03b0/integration-tests/src/test/java/org/jboss/narayana/blacktie/jatmibroker/xatmi/KMPMatch.java

d) http://blog.csdn.net/v_july_v/article/details/7041827

e) http://www.cnblogs.com/yjiyjige/p/3263858.html

f) https://github.com/kinshuk4/AlgorithmUtil/blob/78636378d0394f6f675c112a84aee6a417fceb79/src/com/vaani/algo/search/string/KmpStringMatcher.java

g) https://github.com/lightningMan/generate-file-plugin/blob/1363d94e4b92725abb1ae78535c988ba4906e308/src/com/edwin/plugin/parser/TagParserFactory.java

h) http://www.vogella.com/tutorials/JavaPerformance/article.html