International Journal on
Advanced Science
Engineering
Information Technology

Vol. 2 (2012) No. 5
ISSN: 2088-5334

An Empirical Evaluation of BFS, and DFS Search Algorithms on J2ME Platform, and SVG Tiny Parsing on J2ME Platform Using SAX, StAX, and DOM Parsers

Venera Sengirova, Aidana Oralbekova, Nathar Shah

Faculty of Information Technology, Multimedia University, Cyberjaya, 63100 Malaysia
E-mail : nathar.shah@gmail.com

***Abstract*— SVG (Scalable Vector Graphics) Tiny, an XML-based data representation format was used in our Global Train Route Planner J2ME application to render and manipulate train network images. The SVG Tiny format enables the application to be adaptable with any train network map. We compared three parsing models namely DOM (Document Object Model), SAX (Simple API for XML), and StAX (Streaming API for XML) which were used to visualize the images on mobile phone. We present here the result of the runtime performances, and memory footprints of those parsing models. This is a significant study because handheld devices like mobile phones require seamless interactivity (i.e. high performance) with users and an efficient parsing mechanism with less memory footprints. We also empirically investigated two route searching algorithms - graph and matrix based implementation of DFS (Depth First Search), and matrix based BFS (Breadth First Search) – for performance and memory footprints on a J2ME mobile device emulator. We concluded that DOM parser and DFS based on graph implementation are of better performance than the others.**

***Keywords*— XML Parsing, SAX, StAX, DOM, J2ME, SVG Tiny**

I. INTRODUCTION

The XML based Scalable Vector Graphics (SVG) Tiny can be used to render graphics. Not only that, it can be used for zooming, panning, and selecting objects. Due to portable nature of XML documents, this can be used do develop adaptable mobile route planner where different train network maps can be plug and played to find the shortest and cheapest routes. This approach, nevertheless, requires parsing on the mobile device to convert the text based XML document to memory objects accessible by the program. But most mobile devices are typically resource-starved: short in memory, and not having a lot of excess CPU to spend on parsing XML. There are several ways to parse XML document for the J2ME. In this paper, we will compare SAX, StAX, and DOM parsers. In general, there are three types of parsers: push parsers, pull parsers, and model parsers. Push parsers will push information that is of interest as it parses through the entire document. Pull parsers will need to be guided on what to pull next and how to pull it. Model parser on the other hand, parses the document and creates in-memory representation using nested objects

SAX [Michael, 04], a push based parser, will read from beginning to end and generate an event when it encounters an XML entity. The handler attached to an interested event will perform application-specific tasks for the event. This approach does not preserve the structure and content information in memory, thus saving a large amount of memory space. Unfortunately, they lack the ability to random access and are forward access only, which limits their use to a very small scope.

StAX [Michael, 04], a pull parser, gives programmer more control compared to SAX parser. Instead of emitting event while parsing from beginning till end like SAX, StAX allows the next event to be "pulled". This way, once an interested event is obtained, the parsing can stop and the rest of the document need not be processed. This approach is effective for resource constrained mobile devices.

DOM [Michael, 04], a model parser, creates a node object in-memory tree representation for each node that precisely model all the structure and content information of the XML document. Unlike SAX and StAX parsing which traverse hierarchical data linearly, DOM parsing has the full hierarchical representation in-memory thus enables the program to access and manipulate any data randomly using a set of API methods.

Due to resource constrained nature of mobile devices, it is significant to evaluate the performance and memory footprints of different parsing mechanisms, and different implementations of searching algorithms to choose the best that will fit route planning mobile application. Hence, in this paper, we present empirical evaluations of performance and memory footprints of SAX, StAX, and DOM parsers parsing SVG Tiny files (containing train network maps), and also empirical evaluations of different implementations (i.e. matrix based, and graph based) of BFS, and DFS algorithms.

The next section explains in more detail the three parsers and their evaluations from the memory utilization and CPU performance perspectives. The following section 3 then delve into Breadth First Search (BFS) and Depth First Search (DFS) empirical evaluations on a mobile platform from the perspective of memory utilization and CPU performance. The search algorithms are used to find the shortest and cheapest routes. They are implemented in two methods: graph based approach, and matrix based approach. The final section concludes our works.

II. PERFORMANCE AND MEMORY FOOTPRINTS OF DOM, SAX, AND STAX PARSERS

DOM produces many node objects to build a tree object [Nicola, 03] [Zhao, 06]. Each node object stores element name, attributes, namespaces, and pointers to indicate the parent-child-sibling relationship. For example, in figure 1 the node object stores the element name of Path as well as pointers to its parent (SVG), child (id, link_to, price, stroke, d), and siblings (Text, and Rect).

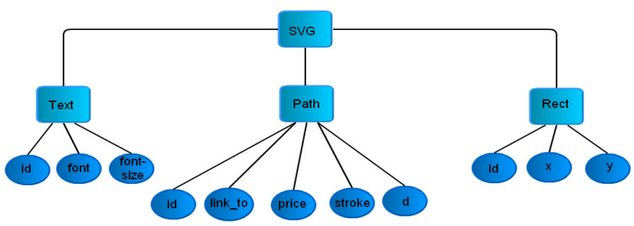


Fig.1 DOM tree representation of SVG document

SAX and StAX [Java, 05] parsers on the other hand, associate different objects with different events and do not maintain the structures among objects. For example, in figure 2, the start element event is associated with three String objects and an Attribute object. The end element event is similar to the start element event without an attribute list. The attribute list's link_to, and price are custom attributes referring to connecting stations, and their traveling costs respectively.

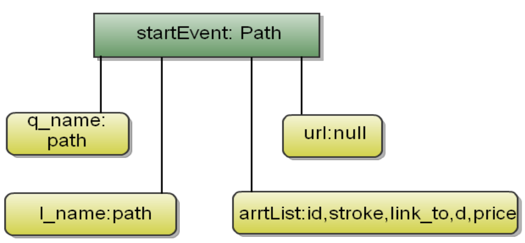


Fig.2. SAX and StAX representation of Path Node

We used kXML parser, a light footprint parser, to implement the Pull (StAX) and Model (DOM) parsing techniques. For SAX, we utilized JSR172 (Java API for XML Processing).

SAX and StAX interlace parsing and access, so the application can access partial data before parsing is complete. Because the objects associated with events can be destroyed regularly, memory usage does not grow with document size [Zhao, 06][Java, 05].

SAX adopts the push model, which uses callback functions to report events from the parser to the application [Nicola, 03]. The parser has a loop to continuously check tokens produced from lexical analysis. When it finds a token, the parser invokes a callback function based on the token type such as startElement(..), endElement(..), characters(..).

In contrast, StAX adopts the pull model [Java, 05]. An application in the pull model can skip uninterested events by calling nextEvent(), whereas an application in the push model must handle all events fed from the parser. The pull model does not need to maintain states between callback functions to decide correct actions, making the programming flow more natural and maintainable. A common misconception is that pull parsers are always faster than push parsers because they save effort by skipping uninteresting events. However, numerous studies reveal that this is not always true [Java, 05]. Although the application can skip events by calling nextEvent(), the parser still creates the events sequentially without skipping them. Performance therefore depends on the application needs. If the application has to navigate through the entire document, the pull model has little advantage over the push model, but if it can stop parsing after accessing certain uninteresting data, the pull model is faster.

Comparison of SAX, StAX and DOM parsing algorithms were done using WTK Profiler 2.2 for our mobile application. This profiler is embedded into Wireless Toolkit Emulator and provides memory monitor as well as general CPU performance profiler. Pull parsing executed around 270 millions (refer to Table1) of cycles in total while parsing whole SVG file and outputting the list of stations.

TABLE I
CPU PERFORMANCE OF STAX PARSING TECHNIQUE

Name	Count	Cycles	%Cycles	Cycles ...
<root>	0	0	0	269925262
com.sun.midp.main.Main.main...	0	18273	0	138551903
kXMLDemo_pull.startApp	1	311	0	138529261
kXMLDemo_pull.initMIDlet	1	1742	0	138528950
kXMLDemo_pull.beginParse	1	638495	0,2	136800446
com.sun.midp.lcdui.DefaultEventHandler$QueuedEventH...	0	131221531	48,6	131221531
org.kxml.parser.XmlParser.peek	808	1090594	0,4	129357046
org.kxml.parser.XmlParser.parseSpecial	343	222271	0	124205847
org.kxml.parser.XmlParser.parseStartTag	231	8189355	3	121991180
org.kxml.parser.XmlParser.read	808	255370	0	101702236
org.kxml.parser.XmlParser.readText	2039	24621913	9,1	54688502
kXMLDemo_pull.parseAddressTag	1	730609	0,2	36355615
org.kxml.parser.XmlParser.readName	2037	12151734	4,5	31252632
org.kxml.parser.XmlParser.readChar	59520	21874785	8,1	30424265
org.kxml.parser.XmlParser.peekChar	123581	18007598	6,6	18929785
org.kxml.parser.StartTag.<init>	231	3369264	1,2	12324025
java.lang.StringBuffer.toString...	2382	9792183	3,6	9792183

Based on the Figure 3, the current amount of memory used by the mobile application for Pull parsing was around 215Kbytes. The maximum amount of memory used in Pull parsing algorithm since program execution begun was around 477Kbytes. Maximum memory usage is denoted in the graph by a broken red line.

In case of SAX parsing, total amount of cycles is around 195 millions of cycle (refer to Table 2), which is less than StAX's total amount of cycles (270million). It means that, in case of SAX parser CPU performance is better than in StAX.

When comparing memory usage between SAX and StAX parsing algorithms, maximum amount of memory used in SAX parsing algorithm is around 493Kbytes (Refer to Figure 4) which is a bit more than in StAX.

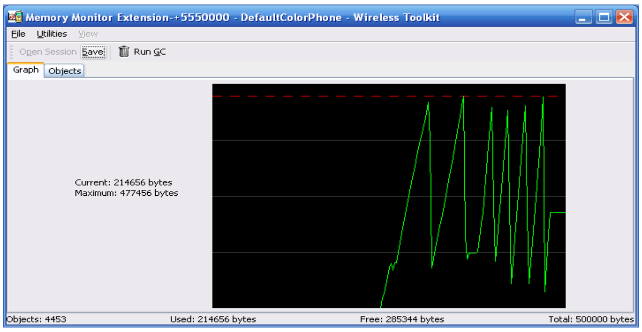


Fig.3. Memory Monitor Graph for PULL parsing

TABLE II
CPU PERFORMANCE OF SAX PARSING TECHNIQUE

<root>	0	0	0	195468906
com.sun.midp.lcdui.DefaultEventHandler$QueuedEventHandler.r...	0	120063779	65,5	120063779
com.sun.midp.main.Main.main...	0	19758	0	67266857
hello.SaxParsing.startApp	1	240	0	67243063
hello.SaxParsing.initMIDlet	1	1290	0	67242823
hello.SaxParsing.SAXParse	1	7823	0	66066326
com.sun.ukit.jaxp.Parser.parse...	2	2184249	1,1	65897293
com.sun.ukit.jaxp.Parser.parse	460	1249198	0,6	63712826
com.sun.ukit.jaxp.Parser.ent...	2	61268326	31,3	61268326
javax.microedition.lcdui.Display.setCurrent...	2	1279304	0,6	1279304
hello.SaxHandler.startElement	231	78138	0	791456
java.lang.String.equalsIgnoreCase...	229	713206	0,3	713206
com.sun.ukit.jaxp.Parser.bflash	114	37571	0	255757
hello.SaxHandler.characters	338	115686	0	243311

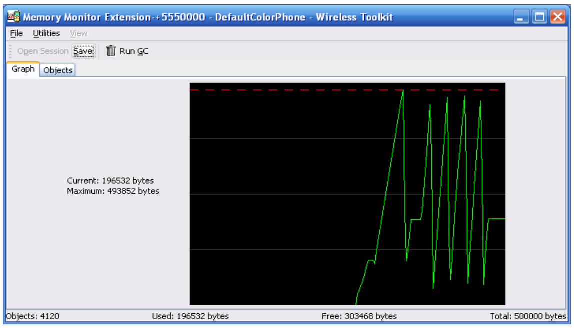


Fig.4. Memory Monitor Graph for SAX parsing

DOM can access data only after parsing is complete – that is, when the loop inside the parser program can draw no more tokens from lexical analysis to construct the tree. A large document will significantly delay data access [Nicola, 03] [Zhao, 06]. Moreover, the two models' long-lived data representations make memory usage grow with document size, which is undesirable for streaming.

In case of our mobile application, DOM parsing executed around 39 millions of cycles in total (refer to Table 3) while parsing whole SVG file and outputting the list of train stations.

TABLE III
CPU PERFORMANCE OF DOM PARSING TECHNIQUE

<root>	0	0	0	38609118
com.sun.midp.lcdui.DefaultEventHandler$QueuedEventHa...	0	31947423	82,7	31947423
com.sun.midp.main.Main.main...	0	1335	0	6469289
kXMLDemo_dom.startApp	1	30	0	6464213
kXMLDemo_dom.initMIDlet	1	357	0	6464183
kXMLDemo_dom.beginParse	1	73439	0,1	6092237
org.kxml.kdom.Document.parse	1	1102	0	5963866
org.kxml.kdom.Node.parse	232	41225	0,1	5962749
org.kxml.kdom.Element.parse	231	23873	0	5723067
org.kxml.parser.XmlParser.peek	1039	54854	0,1	5587109
org.kxml.parser.XmlParser.parseSpecial	343	10265	0	5332852
org.kxml.parser.XmlParser.parseStartTag	231	254248	0,6	5228718

Based on results from Table 1, 2, and 3, DOM parsing algorithm is 5 times faster than SAX parsing (195/39 = 5), and DOM is 7 times faster than StAX(270/39 = 7).

But in case of memory usage, DOM definitely consumes more memory than previous parsers (refer to Figure 5). Maximum amount of memory consumed by DOM parsing algorithm is around 1.9 Mbytes which is 4 times more than in Pull parsing algorithm, and 3.8 times more than in SAX parsing algorithm. But still, the performance in DOM parsing algorithm is much faster than SAX and StAX.

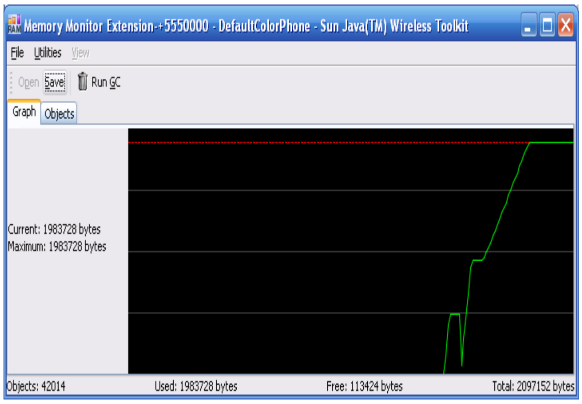


Fig.5. Memory Monitor Graph for DOM parsing

III. DEPTH FIRST SEARCH (DFS) AND BREADTH FIRST SEARCH (BFS) EVALUATIONS

We implemented both DFS and BFS on the mobile environment to find the shortest path and cheapest cost. Two implementations of DFS were done: graph based, and matrix based. For BFS, it was implemented using matrix. In the matrix implementation, two 2D arrays were created to store connectivity between stations and also fares between stations. On the other hand, in the graph implementation, the vertices stored the station info and an adjacent list contained the list of stations linked to the station. The details of the stations, connections, and fares are parsed form the SVG network map. Indent the first line of the second and all subsequent paragraphs. If you use figures, make sure the figures stay within the printing area.

The WTK 2.2 tool was used to obtain the memory footprint and processing performance empirical results. And the profiling includes average of both distant and near stations. The definition of the metrics is as follows:

Current - Current amount of memory used by the application.
Maximum - Maximum amount of memory used since program execution began, shown in the graph by a broken red line.
Objects - Number of objects in the heap.
Used - Amount of memory used.
Free - Amount of unused memory available.
Total - Total amount of memory available at startup.

A. Memory footprint experiment results

DFS algorithm (graph based implementation) evaluation

Fig.6. Memory Monitor Graph for Graph based DFS

Current: 1812352 bytes
Maximum: 1812352 bytes
Objects: 30485
Used: 1812352 bytes
Free: 284800bytes
Total: 2097152 bytes

DFS algorithm (matrix based implementation) evaluation

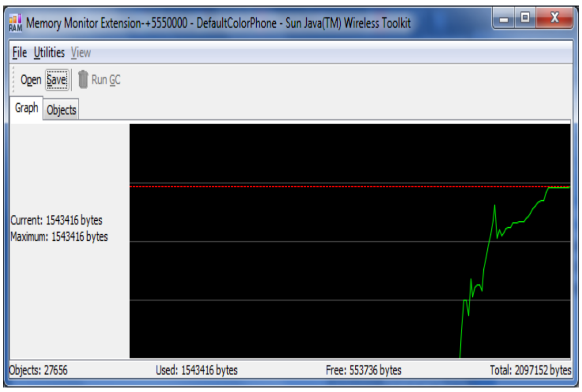


Fig.7. Memory Monitor Graph for Matrix based DFS

Current: 1543416 bytes
Maximum: 543416 bytes
Objects:27656
Used: 543416 bytes
Free: 553736 bytes
Total: 2097152 bytes

B. BFS algorithm (matrix based implementation) evaluation

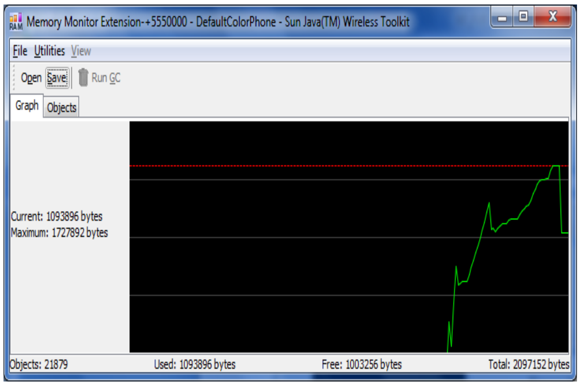


Fig.8. Memory Monitor Graph for Matrix based BFS

Current: 1093896 bytes
Maximum: 1727892 bytes
Objects:21879
Used: 1093896 bytes
Free: 1003256 bytes
Total: 2097152 bytes

C. CPU performance

DFS algorithm (graph based implementation) evaluation

Whole application:
Cycles with children: 16361702
Cycles without children: 100
Graph initialization:
Cycles with children: 1943866
Cycles without children: 11.8
Graph.DFS:
Cycles with children: 468583
Cycles without children: 2.8

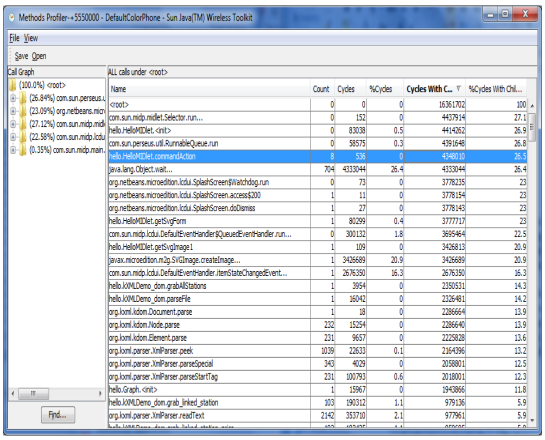
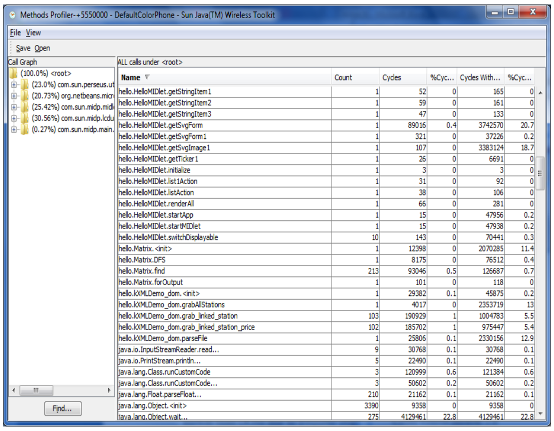


Fig.9. CPU performance table for Graph based DFS

DFS algorithm (matrix based implementation) evaluation



Methods Profiler-+5550000 - DefaultColorPhone - Sun Java(TM) Wireless Toolkit

File View

Save Open

Call Graph

(100.0%) <root>
(23.0%) com.sun.perseus.ut
(20.73%) org.netbeans.micr
(25.42%) com.sun.midp.midl
(30.56%) com.sun.midp.lcdu
(0.27%) com.sun.midp.main.

Find...

ALL calls under <root>

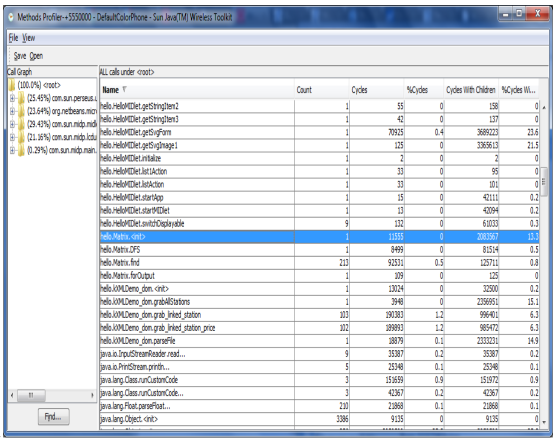
Name	Count	Cycles	%Cyc...	Cycles With...	%Cyc...
hello.HelloMIDlet.getStringItem1	1	52	0	165	0
hello.HelloMIDlet.getStringItem2	1	59	0	161	0
hello.HelloMIDlet.getStringItem3	1	47	0	133	0
hello.HelloMIDlet.getSvgForm	1	89016	0.4	3742570	20.7
hello.HelloMIDlet.getSvgForm1	1	321	0	37226	0.2
hello.HelloMIDlet.getSvgImage1	1	107	0	3383124	18.7
hello.HelloMIDlet.getTicker1	1	26	0	6691	0
hello.HelloMIDlet.initialize	1	3	0	3	0
hello.HelloMIDlet.list1Action	1	31	0	92	0
hello.HelloMIDlet.listAction	1	38	0	106	0
hello.HelloMIDlet.renderAll	1	66	0	281	0
hello.HelloMIDlet.startApp	1	15	0	47956	0.2
hello.HelloMIDlet.startMIDlet	1	15	0	47938	0.2
hello.HelloMIDlet.switchDisplayable	10	143	0	70441	0.3
hello.Matrix.<init>	1	12398	0	2070285	11.4
hello.Matrix.DFS	1	8175	0	76512	0.4
hello.Matrix.find	213	93046	0.5	126687	0.7
hello.Matrix.forOutput	1	101	0	118	0
hello.kXMLDemo_dom.<init>	1	29382	0.1	45875	0.2
hello.kXMLDemo_dom.grabAllStations	1	4017	0	2353719	13
hello.kXMLDemo_dom.grab_linked_station	103	190929	1	1004783	5.5
hello.kXMLDemo_dom.grab_linked_station_price	102	185702	1	975447	5.4
hello.kXMLDemo_dom.parseFile	1	25806	0.1	2330156	12.9
java.io.InputStreamReader.read...	9	30768	0.1	30768	0.1
java.io.PrintStream.println...	5	22490	0.1	22490	0.1
java.lang.Class.runCustomCode	3	120999	0.6	121384	0.6
java.lang.Class.runCustomCode...	3	50602	0.2	50602	0.2
java.lang.Float.parseFloat...	210	21162	0.1	21162	0.1
java.lang.Object.<init>	3390	9358	0	9358	0
java.lang.Object.wait...	275	4129461	22.8	4129461	22.8

Fig.10. CPU performance table for Matrix based DFS

Whole application:
  Cycles with children: 18047977
  Cycles without children: 100
matrix initialization:
  Cycles with children: 2083567
  Cycles without children: 13.3
Matrix.DFS:
  Cycles with children: 76512
  Cycles without children: 0.4

BFS algorithm (matrix based implementation) evaluation

Whole application:
  Cycles with children: 15606421
  Cycles without children: 100
matrix initialization:
  Cycles with children: 2083567
  Cycles without children: 13.3
Matrix.BFS:
  Cycles with children: 81514
  Cycles without children: 0.5



Methods Profiler-+5550000 - DefaultColorPhone - Sun Java(TM) Wireless Toolkit

File View

Save Open

Call Graph

(100.0%) <root>
(25.45%) com.sun.perseus.u
(23.64%) org.netbeans.micr
(29.43%) com.sun.midp.midl
(21.16%) com.sun.midp.lcdu
(0.29%) com.sun.midp.main.

Find...

ALL calls under <root>

Name	Count	Cycles	%Cycles	Cycles With Children	%Cycles Wi...
hello.HelloMIDlet.getStringItem2	1	55	0	158	0
hello.HelloMIDlet.getStringItem3	1	42	0	137	0
hello.HelloMIDlet.getSvgForm	1	70925	0.4	3689223	23.6
hello.HelloMIDlet.getSvgImage1	1	125	0	3365613	21.5
hello.HelloMIDlet.initialize	1	2	0	2	0
hello.HelloMIDlet.list1Action	1	33	0	95	0
hello.HelloMIDlet.listAction	1	33	0	101	0
hello.HelloMIDlet.startApp	1	15	0	42111	0.2
hello.HelloMIDlet.startMIDlet	1	13	0	42094	0.2
hello.HelloMIDlet.switchDisplayable	9	132	0	61033	0.3
hello.Matrix.<init>	1	11555	0	2083567	13.3
hello.Matrix.DFS	1	8499	0	81514	0.5
hello.Matrix.find	213	92531	0.5	125711	0.8
hello.Matrix.forOutput	1	109	0	125	0
hello.kXMLDemo_dom.<init>	1	13024	0	32500	0.2
hello.kXMLDemo_dom.grabAllStations	1	3948	0	2356951	15.1
hello.kXMLDemo_dom.grab_linked_station	103	190383	1.2	996401	6.3
hello.kXMLDemo_dom.grab_linked_station_price	102	189893	1.2	985472	6.3
hello.kXMLDemo_dom.parseFile	1	18879	0.1	2333231	14.9
java.io.InputStreamReader.read...	9	35387	0.2	35387	0.2
java.io.PrintStream.println...	5	25348	0.1	25348	0.1
java.lang.Class.runCustomCode	3	151659	0.9	151972	0.9
java.lang.Class.runCustomCode...	3	42367	0.2	42367	0.2
java.lang.Float.parseFloat...	210	21868	0.1	21868	0.1
java.lang.Object.<init>	3386	9135	0	9135	0

Fig.11. CPU performance table for Matrix based BFS

From observations above, the DFS based on graph performed better than the DFS based on matrix. However, the BFS based on matrix gave the best overall result. In the aspect of memory footprint, matrix implementation left less footprint compared to graph implementation of data structure.

IV. CONCLUSION

Based on the evaluation done in previous section, we can conclude that SAX and StAX do not maintain long lived structural data and are limited to sequential access. Memory consumption depends on location of particular element in the document. In order to modify, the application must buffer the entire document before it can alter the document. SAX and StAX thus do not have an advantage in terms of memory consumption as they do in streaming applications. For this reason, SAX and StAX are typically used for forward-only applications or simple modifications.

In contrast to SAX and StAX, DOM maintains parent-child-sibling information in their long-lived structural data. Preparing this data incurs more overhead, but the simple-to-navigate tree ease access. DOM is better because of its modification capability. DOM is more suitable for massive and frequent updates. It is possible to add or delete a node to or from the DOM tree by simply manipulating the pointers between tree nodes. The modified tree is then ready for further updates.

From figure 12 it is seen that the DOM parsing algorithm consumes much more less CPU power while SAX and StAX take more time to parse XML document. From figure 13, we can conclude that SAX and StAX are appropriate for applications with extremely restrictive memory but not for backend- forth access or modification.

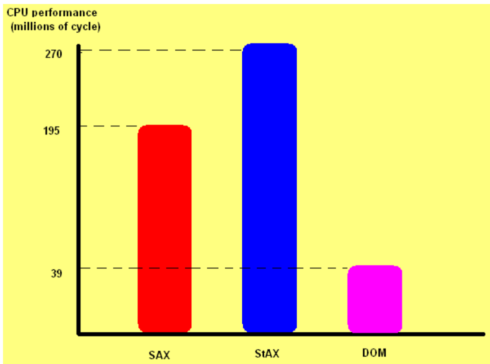


Fig.12. CPU performance diagram

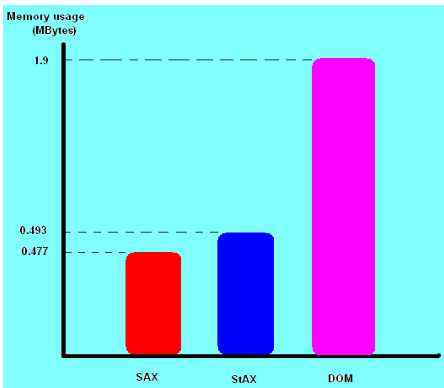


Fig.13. Memory usage diagram

For our mobile application we used DOM parsing technique, because of the need for fast CPU response. Even if it consumes a lot of memory, it is worth to say that mobile phone holders are keen to be impatient when it comes to fast response of the application. As in our application we are not using large file to parse, the memory concerns should not be so restrictive.

As for the searching algorithm, DFS algorithm implemented based on graph data structures were chosen. This choice was made because user would prefer faster output than memory usage amount.

REFERENCES

[1] Michael Juntao Yuan, Enterprise J2ME: developing mobile Java applications, Prentice Hall, 2004

[2] M. Nicola and J. John, “XML Parsing: A Threat to Database Performance,” Proc. 12th Int’l Conf. Information and Knowledge Management (CIKM 03), ACM Press, 2003, pp.175-178.

[3] L. Zhao and L. Bhuyan, “Performance Evaluation and Acceleration for XML Data Parsing,” Proc. 9th Workshop Computer Architecture Evaluation Using Commercial Workloads (CAECW 06), 2006

[4] Java Web Services Performance Team, Streaming APIs for XML persers, 2005 Retrieved from http://java.sun.com/performance/reference/whitepapers/StAX-1_0.pdf