A Client-side Intelligent Paging Algorithm Based on JavaScript

Yinhuang Le
Lab of Modern Educational Technology
University of Science and Technology of China
Hefei, China
yinhuang@mail.ustc.edu.cn

Jiahui Qi, Min Wu
Lab of Modern Educational Technology
University of Science and Technology of China
Hefei, China

***Abstract*—In this paper, a client-side intelligent paging (CIP) algorithm based on JavaScript technique is proposed. CIP algorithm can intelligently turn a large-scale HTML document with an intricate DOM tree into more manageable pages with relatively simple structure of DOM tree. There are two main co-dependent procedures in CIP: self-corrective element node splitting and self-adaptive text node splitting. Experiments indicate that the loading time is reduced by 500% at most and not a single page overflows. Thus, CIP can provide significant improvement in large-scale HTML document paging.**

Keywords-intelligent paging; manageable pages; self-correct; self-adaptive; large-scale HTML document;

I. INTRODUCTION

With the increasing of massive open online courses and a variety of convenient mobile terminals, a great demand of access of online textbooks and e-books is observed in the incoming digital reading era. Meanwhile, manufacturers such as Apple, Google, and Amazon launch their new pad to introduce more interactive, useful and delicate e-books with fascinating page-flipping effects, rather than just scrolling a whole book [1]. These e-books are HTML-based and they are essentially split web pages. Thus it turns out that the paging algorithm is the core part of making e-books, which directly affects the user experience [2].

Paging algorithm makes up the defects of the popular PDF-based e-books, such as tedious scrolling, static text-only, and image without any interaction. Nowadays, most client-side paging algorithms are based on part of HTML5 technique [3]: either CSS, JavaScript, or the combination of two. In essence, they are all web-dependent technologies. In this paper, JavaScript-based paging technique is adopted to implement the paging algorithm.

A paging algorithm is used to make a HTML document, which is large-scale with a considerable amount of DOM nodes, into separate consecutive pages, which have relatively small number of DOM nodes that with simple structure. Two sub-processes, including breaking the original DOM tree and reconstructing many new small compact trees, cost most of the running time of paging algorithm. Thus, for a better user experience, more and more experiments and advancements of paging algorithms begin to emerge.

Recently, a variety of paging algorithms are widely implemented in e-books for its high convenience. However, there are many disadvantages of current common paging algorithms, such as HTML truncation, loading speed, page consistency, etc. All these factors would probably lead to a really awkward and uncomfortable user experience. Thus, in this paper, a more intellectual, effective, and faster algorithm is proposed. The experiment and test are also provided.

II. CIP ALGORITHM

Although with the fast developments of the front-end technology, there are still really few paging algorithms depending only on the client-side browser. In this paper, the client-side intelligent paging (CIP) algorithm is put forward.

CIP is regarded as an intelligent algorithm mainly because that it is self-corrective when it overflows a page. This should be ascribed to its design, including process design and data structure design.

A. Process Design

The process of CIP is as follows. There is an original HTML document named *source*, which has a large-scale DOM tree, needing to be divided into fragments that are consecutive but not always have the same size. Subsequently, fragments are thrown into runtime-created containers named *temp* in sequence. Giving a pre-defined page height as threshold, if the height of a *temp* exceeded the threshold, the *temp* is considered to be full. Therefore another *temp* would be produced to contain fragments. Similar process runs one after another until *source* goes empty. Finally, well supported by the structure that *temp* provides and sufficiently filled by the contents that get from the fragments of *source*, corresponding pages come into being. To be more clearly, figure 1 depicts a more direct view of the process.

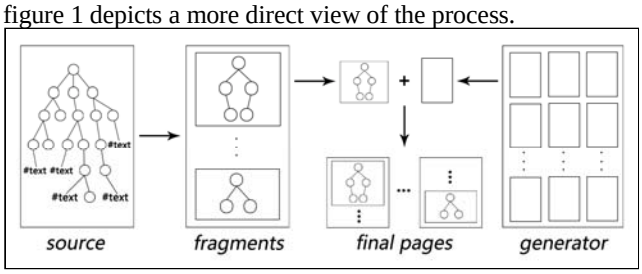

Figure 1. The main process of CIP algorithm

As it is simply shown above, the whole process is exactly like a situation that always happens on the production line, say, coca cola. Coke incessantly comes out of source; meanwhile, containers like bottles are constantly pushed to fill coke until the source dries out. Thus, people can enjoy coca cola. Indeed, CIP algorithm is far more complicated. Having acquainted a macro view of CIP, it is turn to take a

deeper step internally to understand how it works in a micro perspective. The specific steps are as follows.

- Step 1: Load the original HTML source file. Create a pointer named *src*. Let *src* point to source file. And Create a pointer named *targ*.
- Step 2: Create a new container and let *targ* point to the new container.
- Step 3: Take out the first child of *src*, name it *first*.
- Step 4: Append *first* into current container. As a result, the height of container would grow.
- Step 5: Determined by some comparison strategy of height, which will be described after, if the current container is not full, continue with execution of Step 2 and Step 4. Else if it does overflow, go to Step 6.
- Step 6: Check the node type of current *first*, if it is an element node, go to Step 7. Else if it is a text node, go to Step 8.
- Step 7: Take the current *first* from the current container back to the current source. Depending on the internal structure of *first*, if *first* can be split, clone the most peripheral structure of *first*, name it *shadow*, and append *shadow* to the current container. Then shift the pointer *src* and *targ* respectively to *first* and *shadow*, which means both pointers go a step deeper in both source and container. After that, continue with Step 3, 4, 5, 6. Else if *first* cannot be split, continue with Step 2, 3, 4, 5, 6.
- Step 8: Create a one-dimensional array variable named *splitPoints* to store the total split points that *first* may have. Apply binary search to *spiltPoints*, if an appropriate split point can be found, take the text from the beginning of *first* to where the appropriate split point stands, use them to replace the *first*. Then let *src* point to the source file, and another loop will go with Step 2, 3, 4, 5, 6. Else if none split point was found, let *src* point to the source file and continue with Step 2, 3, 4, 5, 6. When there is no more content in the source file, this step ends, which in turn brings the whole paging process to an end.

B. Data Structure Design

As mentioned above, along with the process of CIP, the algorithm deals with a DOM tree, whatever scale it is, all the time. Obviously, CIP is chiefly based on tree-form data structure. All data is organized in a DOM tree. The source file is a gigantic DOM tree while fragments broken from the source have relatively small size DOM tree.

By using tree-form data structure, traversing all data becomes enormously conveniently, especially for the shift of two pointer: *src* and *targ*, which play a significant role in the process of CIP. To demonstrate their importance, an example of producing fine-grain DOM trees from a coarse-grain DOM tree is provided as follows. For simplicity, the layers of a DOM tree are confined to 3 and the number of nodes is limited to 10. In addition, each node has different size and complexity. Figure 2 to 5 depict a more detailed scene how fragments are generated from a DOM tree by the method of depth-first search (DFS) [4].

Initially, *src* points to the root of a DOM tree, and *targ* points to a newly-created container, as depicted in Fig 2. The dotted oval frame indicates the first sub-tree to be moved.

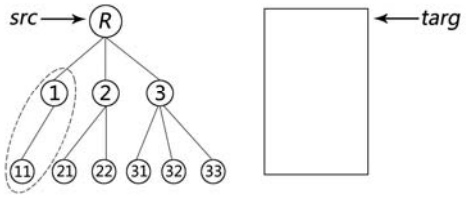

Figure 2. The first step of DOM tree splitting based on DFS

Fig. 3 shows the scene after filling the first child of *src*, which is the sub-tree that the dotted oval frame in Fig. 2 enclosures, into *targ*. Suppose that the first child can be perfectly filled in, therefore *src* and *targ* will just stay where they were.

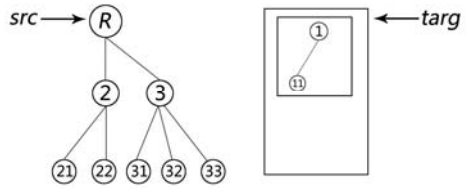

Figure 3. The second step of DOM tree splitting based on DFS

As CIP goes to next loop, Fig. 4 reveals what is going on at this stage. Node 2 is now the current first child. The sub-tree of Node 2 is too big to squeeze into the current container as a whole. Thus, the tree-form structure of the sub-tree is cloned and thrown into *targ*. Meanwhile, *targ shifts to point to* the empty clone and *src* goes a layer deeper to point to Node 2.

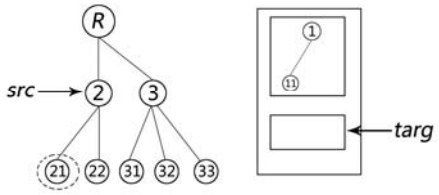

Figure 4. The third step of DOM tree splitting based on DFS

When the algorithm comes down to this step, assume that it is checked that only Node 21 can be taken out, then the state will be exactly what Fig. 5 presents below. A new container has been created and *targ* points to it. At the same time, *src* is set to point the root node again.

Up to this point, the most complicated splitting in element node level has been handled. By flexibly shifting *src* and *targ*, it is clearly that CIP can be clever enough to cope with any splitting task in element node level. Thus, CIP is considered to be self-corrective without taking a wrong

direction in a macro scenario when paging. Moreover, in a micro scenario, there is a co-dependent mechanism along with the self-corrective one. That is the self-adaptive splitting in text node level, which will be described in the next section.

Overall, the design of CIP, in essence, is heavily involved with manipulations of DOM tree and process of recursively breaking DOM tree as well as reconstructing it.

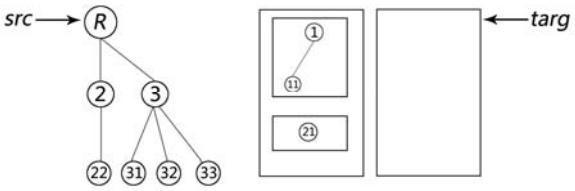

Figure 5. The fourth step of DOM tree splitting based on DFS

III. IMPLEMENTATION MODEL

In the previous section, the whole process and data manipulations of CIP algorithm have been amply elaborated. Where there is an idea, there is a model. Hence, in this section, the implementation model is proposed to achieve client-side intelligent paging program.

There are two modules for the implementation model:

- The main module: *fillPages( )*.
- The auxiliary module: *makePage( )*.

As it is mentioned before, the relationship between the two modules is quite simple. The following Fig. 6 draws a clearer view about this.

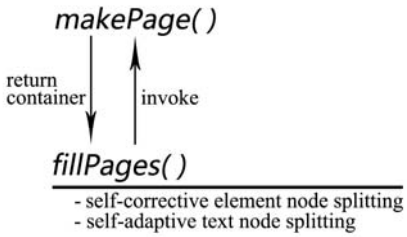

Figure 6. Relationship between two modules

The auxiliary module is really straight forward. It is invoked when *fillPages( )* needs a new container. Then a container with a nested two-layer is created and return to *fillPages( )*. Pseudo codes of *makePage( )* and *fillPages( )*, which can be terrifically useful for understanding the whole idea how CIP acts intellectually, go as follows.

fillPages(src, targ, page)

```
Input:
    Enter a number to set preHeight
Initialization:
    Set source ← the original HTML file;
    Set firstPage = makePage( );
    Set src = source; targ = firstPage; page = firstPage;
Iteration:
 1: if source is empty
 2:   then done and stop
 3: else
 4:   first ← src.firstChild
 5:   append first to targ
 6:   if (page.Height <= preHeight)
 7:     then fillPages(src, targ, page)
 8:   else
 9:     if first is an element node
10:       then prepend first back to src
11:       if first can be split
12:         then shadow ← clone only structure of first
13:         append shadow to targ
14:         fillPages(first, shadow, page)
15:       else
16:         targ, page ← makePage( )
17:         fillPages(source, targ, page)
18:     if first is a text node
19:       then txt ← all texts of first
20:       splitPoints[0] ← 0
21:       for i ← 0 to txt.length do
22:         if txt[i] is blank, tab, enter, or Chinese char
23:           then splitPoints[j ← 1] ← i
24:           j ← j+1
25:           i ← i+1
26:       End for
27:       len ← splitPoints[ ].length
28:       p ← ⌈len/2⌉
29:       index ← 0
30:       while p ≠ 1 do
31:         index ← splitPoints[p-1]
32:         first ← texts from txt[0] to txt[index]
33:         if (page.Height >= preHeight)
34:           then len ← p
35:           p ← ⌈p/2⌉
36:         else p ← p+⌈(len-p)/2⌉
37:       End while
38:       if (index>=txt.length)
39:         then remove src
40:         else if (index>0)
40:           then keep first in targ
41:         else remove first from targ and add txt to src
42:       End if
43:       targ, page ← makePage( )
44:       fillPages(source, targ, page)
45:   End if
46: End if
```

Figure 7. The pseudo codes of *fillPages( )*

```
makePage( )
1: newPage ← create a DOM element as a container
2: return newPage
```

Figure 8. The pseudo codes of *makePage( )*

Most of the pseudo codes described above have been discussed a lot in process design and data structure design, including the main process and self-corrective splitting of element nodes, respectively. There is still one more vital part hasn't been covered yet. That is self-adaptive splitting of text nodes. The following interprets how CIP adjusts itself to text node splitting by converging to the same solution.

In most cases, text nodes are quite often buried as the deepest nodes in a DOM tree, called leaf nodes. When the splitting is applied to a text node, it probably implies that CIP algorithm is almost about to create a new container, since it knows there is no enough space in current container for an element node. Whether an appropriate split point can be found or not, a new container will be produced. It is not hard to catch up with this idea. Suppose a split point is observed. In this situation, the text would be cut into two parts. After the former part filled into the current container, it would be then full. Thus the latter part of the text would have to wait for a new container to hold it. Otherwise no split point suggests that the current container is already full, even a single word. Apparently, in such circumstances, a new container is absolutely necessary.

Dealing with identifying a split point, the binary search method is introduced [5]. Exactly as the pseudo codes have already clarified from line 27 to 37, the searching process works as follows:

- Step 1: Reduce the length of array that contains all split points by half.
- Step 2: Search in the first half. If *page* overflowed, reduce the length by half again and search until a split point is found or there is none to be found. Otherwise search in the second half, in this case, a split point is affirmatively existing and will be found.

Actually during the search, an indicator called index is created and returned after the binary search iteration to instruct subsequent executions, which determines the final result of text node splitting.

So far, the whole idea and all the details about CIP algorithm have been thoroughly covered. In next section, the use of CIP algorithm for a project will be presented.

IV. Experiment and Test

Having gone through CIP design and implementation model, it is high time to bring the model into reality. As it is mentioned before, CIP is a client-side algorithm. So the experimental environment is based on client-side. More specifically, CIP is programed with JavaScript and runs on various browser platform. In this section, the running test of CIP algorithm and comparison with some other paging algorithm are presented.

For the sake of simplicity, our experiment didn't involve any specific CSS effects. The performance of CIP is evaluated in Google Chrome 23.0.1. The platform is a 3.00GHz Intel Dual-Core machine running Windows 7 premium with 4GB RAM. The test set is a HTML page of 96KB file size with 3524 nodes. The loading time, the number of nodes after splitting, and the correctness are adopted as measurements of CIP performance. To be more scientific, the measurements are averaged over ten runs. There are still really very few client-side paging algorithms widely-used in browser. Thus in this experiment, a quite famous paging algorithm from [6] is set as the experimental control group. The results are provided in the following table.

TABLE I. PERFORMANCE OF DIFFERENT PAGING ALGORITHMS

	Measurements		
	Node numbers after paging	***Loading time/ms***	***Overflow***
CIP	3960	1620	No
[6]	3964	6210	Yes

As it shows in the table, CIP performs much better than the famous paging algorithms chosen, in both the speed and user experience area.

V. Conclusion

This paper presents a client-side paging (CIP) algorithm that can intelligently detect when and where is most appropriate to split a large-scale HTML document. Much of the attention is paid to two co-dependent procedures: self-corrective splitting in element node level and self-adaptive splitting in text node level, which remarkably contribute to the intelligence of CIP algorithm. Moreover, CIP is applied to split a huge HTML document into regularly compacted pages that make an interactive and well-paged e-book. In addition, related experimental results are provided to explain the advantages of CIP.

Acknowledgment

This research is supported by the Humanity and Social Science Research Foundation of Ministry of Education of China, 2010 (No. 10JDGC015).

References

[1] Joaquín Cubillo, Sergio Martín, and Manuel Castro, "New technologies applied in the educational process," Proc. IEEE Symp. Global Engineering Education Conference (EDUCON), IEEE press, May. 2011, pp. 575-584, doi: 10.1109/EDUCON.2011.5773195.

[2] Zhu Jihong and Yu Yonghai, "The influence of personal reading experience on the design of digital books," Proc. IEEE Symp. Management and Service Science (ICMSS), IEEE Press, Oct. 2009, pp. 1-2, doi: 10.1109/ICMSS.2009.5301441.

[3] Wikipedia contributors, "HTML5," Wikipedia, The Free Encyclopedia, http://en.wikipedia.org/wiki/HTML5, 2011.

[4] Wikipedia contributors, "Depth-first search," Wikipedia, The Free Encyclopedia, http://en.wikipedia.org/wiki/Depth-first_search, 2010.

[5] National Institute of Standards and Technology contributors, "Binary search," NIST, http://xlinux.nist.gov/dads/HTML/binarySearch.html.

[6] github contributors, "Columnizer-jQuery-Plugin," adamwulf, github, https://github.com/adamwulf/Columnizer-jQuery-Plugin/, 2012.