International Journal of Computer Applications (0975 – 8887)
Volume 4– No.11, August 2010

Main Content Extraction from Detailed Web Pages

Mohsen Asfia
Islamic Azad University, Science & Research Branch
Department of Computer Engineering
Tehran, Iran

Mir Mohsen Pedram
Tarbiat Moallem University
Department of Computer Engineering
Karaj, Tehran, Iran

Amir Masoud Rahmani
Islamic Azad University
Science & Research Branch
Department of Computer Engineering
Tehran, Iran

ABSTRACT

As we know internet detailed web pages contains information which are not considered as primary content such as advertisements, headers, footers, navigation links and copyright information. Also information on web pages such as comments and reviews are not preferred by search engines to index as informative content, thereby having an algorithm to extracts only main content could help better quality on web page indexing. Almost all algorithms have been proposed are tag dependent means they could only look for primary content among specific tags such as <TABLE> or <DIV>. The algorithm in this paper simulates a web page user visit and how the user finds the main content block position in the page. The proposed method is tag independent and has two phases to accomplish the extraction job. First it transforms input DOM tree obtained from input HTML detailed web page into a block tree based on their visual representation and DOM structure in a way that on every node it will have specification vector, then it traverses the obtained small block tree to find main block having dominant computed value in comparison with other block nodes based on its specification vector values. The introduced method doesn't have any learning phases and could find informative content on any random input detailed web page. This method has been tested in large variety of websites and as we will show, it gains better precision and recall based on other compared method K-FE.

Keywords

Web mining, Noise elimination, Informative content, Information retrieval, Information extraction.

1. INTRODUCTION

A web page structure and layout varies depend on different content type it will represent or the tastes of designer styling its content. Thereby main content position or the main tag containing main content differs in variety of websites. Even there might be some content in page view that are besides each other but actually in DOM tree they are not in the same level and same parents, so finding the main content in this area that doesn't follow any specific rules for arranging and positioning elements needs complicated and costly algorithms.

Algorithms that could simulate a user visiting a website, in high probability could find informative content as result because in most cases actual users in internet could find the area of the main content. But which specifications and structures could help an algorithm to find main content?

Figure 1 shows VIPS algorithm [5] uses visual cues to produce content structure from DOM structure and with this content structure it fills the gap between DOM structure and the conceptual structure of the webpage. The algorithm uses obtained content structure and tries to simulate how actual user finds a main content by blocking the page based on structure and visual delimiters. The blocking result is satisfactory but the algorithm does many loops to reach its desire granularity.

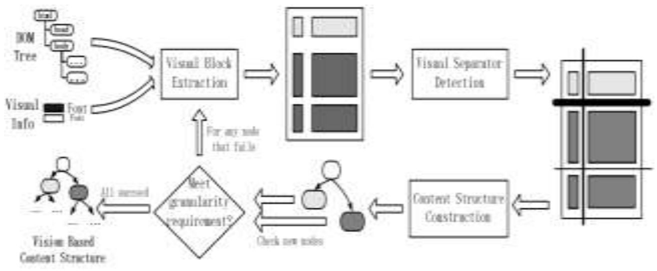

DOM Tree
Visual Info
Font
Visual Block Extraction
Visual Separator Detection
Content Structure Construction
For any node that fails
Meet granularity requirement?
All succeed
Check new nodes
Vision Based Content Structure

Fig. 1. VIPS algorithm process [5].

CE [11] considers all detailed pages of a website as pages with the same class. It runs a learning phase with two or more pages as its input and finds the blocks that their pattern repeats between input pages and marks them as non-informative blocks then stores them in storage. These non-informative blocks are mostly copyright information, header, footer, sidebars and navigation links. Then when we use CE algorithm in actual world it first eliminate non-informative patterns from the structure of its input pages based on the stored patterns in its storage for specific class of input pages. Finally from the remaining blocks in the page it will return the text of block containing the most text length. CE needs a learning phase so it couldn't extract the main content from random one input web page.

FE [11] extracts the text content of a block that has the most probability of having text so it will work fine in web pages that text content of main content dominates other types of content. In addition FE could return just one block of the main content, so [11] proposed K-FE that returns k blocks with high probability of having the main content. Algorithm steps of K-FE and FE are the same except the last part. In K-FE the algorithm final section, sorts the blocks depends on their probability then it uses k-means clustering and takes high probability clusters.

So the proposed paper intends to introduce an algorithm which could extract main content that is not necessarily the dominant content and without any learning phase, with one random page and by using visual cues to simulate user page visit and block the page based on it and gains higher precision.

Next section demonstrates the proposed algorithm in this paper in two main phases, first block tree construction, and then it finds main block from the block nodes in the computed block tree.

2. PROPOSED ALGORITHM

Figure 2 shows the proposed algorithm called Visual Clustering Extractor (VCE). It gets DOM tree of input web page as its input and returns the informative content block as its output.

```
Algorithm 1: Visual Clustering Extractor
Input: DOM tree of an Html page
Output: Primary Content Block
begin
        GetBlockTree(DOM tree, out GeneralParameters,  out BlockTree)
        {Final computation for the averages of GeneralParameters }
        FindMainBlock(BlockTree, GeneralParameters,  out CandidateBlocks, out
                     MainBlock)
        {Output: MainBlock}
end
```

Figure 2. VCE algorithm general look.

The first line of the algorithm use *GetBlockTree* sub-algorithm and sends the DOM tree as its input. The *GetBlockTree* sub-algorithm recursively traverses the input DOM tree in pre-order and returns *block tree* and *general parameters* as its output. Each node in block tree clusters one or more nodes from DOM tree so the number of elements in block tree is significantly lower than the number of nodes in DOM tree. Each node in block tree has specification vector that we will use it to find the main block later. General parameters contain general total value of specifications for *text*. *Link density*, *width* and *height* and we can use these values to compute average values before starting to find the main block. In *Section 2.1 GetBlockTree* sub-algorithm will introduce in detail.

As depicted in Figure 2 the third line of VCE algorithm is *FindMainBlock* sub-algorithm that traverses obtained block tree from *GetBlockTree* sub-algorithm pre-order and based on general parameters it returns the collection of candidate blocks and finally the main block. *Section 2.2* will demonstrate *FindMainBlock* sub-algorithm in detail.

2.1 Constructing Block Tree

This section will introduce the process of block tree construction through *GetBlockTree* algorithm Figure 3. This algorithm recursively loops through input DOM tree and it produces block tree as final result and meanwhile it flags the blocks such as comment blocks which their patterns repeats in unordinary manner and they should not consider as the main content. In addition besides constructing the block tree, the algorithm append the text of child blocks to their parent so we will have text manipulation just in this section without any need for additional loop for text computation later on the block tree. This sub-algorithm contains sub-sections which will introduce in the next parts.

```
Algorithm 2: GetBlockTree Function (Visual Clustering Extractor Cont)
Input: GeneralParameters, BlockTree
Output: Updated Root element, Updated GeneralParameters, updated BlockTree
Type: Recursive
begin
        PatternCache ⟵ Null
        DetectBlock(RootElement, out BlockTree)
        AddText(RootElement)
        If RootElement has child then
          Foreach Child in RootElement's Children do
               If CheckElementIsValid(Child) then
                  GetBlockTree(Child, out eneralParameters,  out BlockTree)
                  PatternVerifier(Child, Out PatternCache)
                  TextManipulation(Child)
        Else
          PatternVerifier(RootElement, Out PatternCache)

        FinalElementManipulation(RootElement,  out GeneralParameters)
end
```

Figure 3. *GetBlockTree* sub-algorithm gets DOM tree as its input and returns block tree and general parameters as its output.

2.1.1 Why Block

First I define the concept of *block* in this paper. A block is a family of elements with same visual styles and order in DOM tree and their conceptual purpose to appear in the page are the same. The algorithm proposed in this paper tries to first divide the page into some blocks. One thing that we should consider here is why we should make block tree in content extraction algorithms? or further why we should have block? I answer this question with an example. Consider we want to remove the navigation links in sidebar, to accomplish this target we should eliminate the parts of the page which have high link density. If you don't use block then the algorithm will consider each individual node element as a page part and removes all the links in the page even if the links are in the main content, because the link density on each individual link is high. But when you are using block, the algorithm would decide on blocks instead of each individual DOM node. So in our example algorithm will remove the navigation links in side bars and it doesn't remove the links in the main content.

2.1.2 Pattern Cache

Finding repetitive block pattern among child blocks of a block node needs additional loop through them, to prevent this and finding them while we are constructing the sub-block tree of a block node we should have a PatternCache which roles as an array and holds the unique patterns among its child blocks. Later we will use this cache array to count the number of repetitive patterns in child blocks of a specific node.

2.1.3 DetectBlock Method

This method will specify either its input node should form a new block or it should use the block of its parent node in DOM tree. This method makes this decision by checking if the *visual distance* of current node in contrast with its parent is more than a threshold value or not. If its distance was not valid then it would make a new block and the algorithm flags the current DOM node as the parent of new block. If the distance was valid it means we should only add the current element to the block of its DOM parent node. The *visual distance* is the number of differences on their visual styles which are important in this algorithm, such as width, height, font-size, background color, top and left. Actually these visual styles are the parts of a block specification vector.

2.1.4 *AddText Method.*

This method will compute the immediate text of current node and immediate link text of current node and add them to the relevant attribute of its block. Other child nodes could add their text to these texts later. Figure 4 shows the immediate text and immediate link text of a DOM node. The text length and link text length are some of the specifications in a block specification vector.

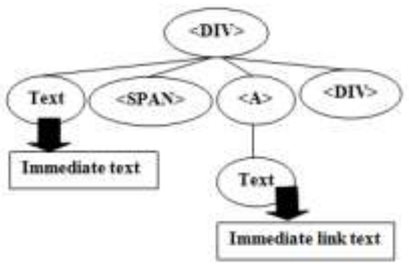


Figure 4. *Immediate text* and *Immediate link text* for a DOM element

2.1.5 *Children Manipulations*

For each DOM children of input element first we do the recursive *GetBlockTree* algorithm. The recursive algorithm make sub block tree for each child node and their subordinating child nodes. Then *PatternVerifier* algorithm runs for the child block with additional input called *PatterCache. PatternVerifier* checks if the current block pattern has any same pattern among its siblings. To accomplish this task, it uses *PatternCache* which contains unique pattern of the current block siblings (**Figure** 5). No of repetitive patterns is a negative specification in a block specification vector.

```
Algorithm 3: PatternVerifier Function (Visual Clustering Extractor Cont)
Input: Element, Pattern Cache holds its sibling's block patterns
Output: Updated number of repetitive block pattern in the parent of input element and
updated Pattern Cache
begin
        {Block pattern set based on root tag types of its children blocks.}
        {Pattern cache has unique patterns.}
        If input element is a block root and
            input element has parent block then
            foreach pattern in PatternCache do
                if the pattern block of input element is equal with current pattern then
                    {Increase the number of repetitive patterns in the parent
                    of input element.}
                    break
            {Add block pattern of input element to pattern cache if there wasn't
            any equal pattern in the foreach loop for it.}
end
```

Figure 5. *PatternVerifier* method

The last process which operates the block of child node is *TextManipulation* method that add the text of child node to its parent if the link density of child node was ok depend on a threshold. Figure 6 depicts more detail for *TextManipulation* method.

```
Algorithm 4: TextManipulation Function (Visual Clustering Extractor Cont)
Input: DOM element
Output: Updated text and link text attributes of input element's parent
begin
        LinkDensity = Link text length of input element / text length of the element
        If visual distance of input element to its parent < threshold OR
            Input element is not parent of any block then
            If Input element Link density < threshold OR
                Input element is a leaf link then
                    {Add its text and text length to its parent.}
            {Add its link text and link text length to its parent.}
end
```

Figure 6. *TextManipulation* method for a DOM element

2.1.6 *Final ElementManipulation Method*

This method updates general parameters such as width, height, text length and link text length, based on updated attributes of current input DOM element. Later the algorithm uses these general parameters for computing average values which can be use for evaluating importance and validity of a block in finding main block section.

So after *GetBlockTree* algorithm finished, we have the block tree and general parameters which can be used for computing general averages. Finally the algorithm sends the block tree and general parameters to *FindingTheMainBlock* sub-algorithm to find the main block.

2.2 Finding the Main Block Based on the Block Tree

This part is the algorithm final round. The *FindMainBlock* sub-algorithm (Figure 7) takes general parameters (general averages) and the block tree. It traverses through the block tree recursively in pre-order manner and for each block it does the following operations:

2.2.1 *Check If Block is Valid*

First it compares the block with general parameters to check if the block met the least conditions to be chosen as candidate for the main block.

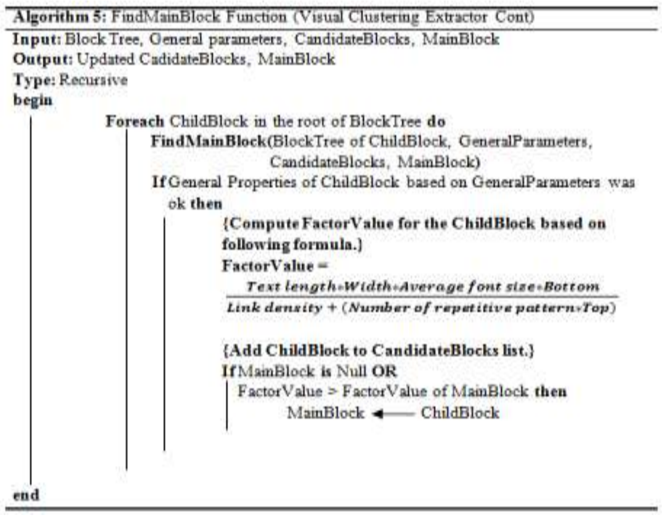
2.2.2 *Compute FactorValue*

Then if the block has those conditions, it computes *FactorValue* for the current block base on its specification vector. The factor value for a block is obtained from the following formula:

$$\text{FactorValue} = \frac{\mathit{Text\ length} * \mathit{Width} * \mathit{Average\ font\ size} * \mathit{Bottom}}{\mathit{Link\ density} + (\mathit{Number\ of\ repetitive\ pattern} * \mathit{Top})} \qquad (1)$$

As we could see in this formula, having more text length, width, average font size and distance from bottom of the page is positive values for a block to be selected as the main block. If a block had repetitive patterns in its children to have higher chance to be selected as *The Main Block* it would be better the distance of the block from the top of the page would be lower and if the number of these rep patterns are lower it would be better. Having higher link density makes it more difficult for a block to be selected as *The Main Block.*

The block that we already compute its *FactorValue* is added to the candidate block list and the algorithm compares its *FactorValue* with the *FactorValue* of *The Main Block*, if the *FactorValue* of candidate block is greater than the *FactorValue* of *The Main Block* then the current block is selected as *The Main Block.*



```
Algorithm 5: FindMainBlock Function (Visual Clustering Extractor Cont)
Input: Block Tree, General parameters, CandidateBlocks, MainBlock
Output: Updated CadidateBlocks, MainBlock
Type: Recursive
begin
    Foreach ChildBlock in the root of BlockTree do
        FindMainBlock(BlockTree of ChildBlock, GeneralParameters,
                      CandidateBlocks, MainBlock)
        If General Properties of ChildBlock based on GeneralParameters was
          ok then
            {Compute FactorValue for the ChildBlock based on
            following formula.}
            FactorValue = $\frac{Text\ length*Width*Average\ font\ size*Bottom}{Link\ density + (Number\ of\ repetitive\ pattern*Top)}$

            {Add ChildBlock to CandidateBlocks list.}
            If MainBlock is Null OR
              FactorValue > FactorValue of MainBlock then
                MainBlock ← ChildBlock
end
```

Figure 7. *FindMainBlock* sub-algorithm, the final round of content extractor

Finally *FindMainBlock* sub-algorithm returns *The Main Block* which has the highest *FactorValue* and the content of this block represented in the output.

3. EXPERIMENTAL RESULTS

In this section we evaluate our algorithm with the dataset in [11] which contains over 5000 pages and compare it with K-FE [11] (because it seems to have better result in comparison with other algorithms), in block-level based on *Block-Precision, Block-Recall* and *Block-F-Measure* factors which are introduced in [11], [3] (Table 1).

Table 1. *Block-Level* comparison between *VCE (proposed algorithm in this paper)* and K-FeatureExtracor [11]

Web sites	Address	b-F-Measure of VCE	b-Recall of VCE	b-Prec of VCE	b-F-Measure of K-FE	b-Recall of K-FE	b-Prec of K-FE
ABC	abcnews.com	1	1	1	1	1	1
BBC	bbc.co.uk	0.98	1	0.98	1	1	1
CBS	cbsnews.com	1	1	1	0.978	0.977	0.98
CNN	cnn.com	1	1	1	0.98	0.98	0.98
FOX	foxnews.com	1	1	1	0.994	0.99	1
FOX23	fox23news.com	1	1	1	1	1	1
MSNBC	msnbc.com	1	1	1	0.95	1	0.92
YAOO	news.yahoo.com	1	1	1	0.974	0.95	1

4. Conclusion

We proposed an algorithm called *VCE* here, which could extract the main content from a random detailed web page. As we saw in section 3 this algorithm gains higher *b-Precision*, *b-Recall* and *b-F-measure* so we gain higher precision in extracted content. The *VCE* algorithm is not dependant on any tag type and it just has an iteration to block its input page while it doesn't have any learning phase. Furthermore it could detect and eliminate comments from the extracted content.

5. REFERENCES

[1] Alberto H. F. Laender, Berthier A. Ribeiro-Neto, Altigran Soares da Silva, Juliana S. Teixeira: A Brief Survey of Web Data Extraction Tools. SIGMOD Record 31(2): 84-93 (2002).

[2] Bing Liu: Web Data Mining. Springer (2007).

[3] C. J. Van Rijsbergen: Information Retrieval. Butterworth-Heinemann (1979).

[4] Deng Cai, Shipeng Yu, Ji-Rong Wen and Wei-Ying Ma: Block Based Web Search. In: Proc. 2004 Int. Conf. on Research and Development in Information Retrieval (SI-GIR'04), Sheffield, UK (July 2004).

[5] Deng Cai, Shipeng Yu, Ji-Rong Wen and Wei-Ying Ma: Extracting Content Structure for Web Pages based on Visual Representation. In: The Fifth Asia Pacific Web Conference (APWeb2003), Springer Lecture Notes in Computer Science (2003).

[6] Deng Cai, Xiaofei He, Ji-Rong Wen and Wei-Ying Ma: Block Level Link Analysis. In: Proc. 2004 Int. Conf. on Research and Development in Information Retrieval (SIGIR'04), Sheffield, UK (July 2004).

[7] Hwanjo Yu, AnHai Doan, and Jiawei Han: Mining for Information Discovery on the Web: Overview and Illustrative Research. In: Intelligent Technologies for Information Analysis, edited by Ning Zhong, Springer-Verlag, invited paper, pp. 135-168 (2004).

[8] Jeff Pasternack, Dan Roth: Extracting Article Text from the Web with Maximum Subsequence Segmentation. In: www '09: proceedings of the 18th international conference on World Wide Web, New York, ny, usa, acm, 971—980 (2009).

[9] Lakshmish Ramaswamy, Arun Iyengar, Ling Liu and Fred Douglis: Automatic Detection of Fragments in Dynamically Generated Web Pages. In: 13th International Conference on the World Wide Web (WWW-2004), pp. 443-454 (2004).

[10] Lan Yi, Bing Liu, and Xiao-Li Li: Eliminating Noisy Information in Web Pages for Data Mining. In: Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD-2003), Washington, DC, USA, August 24 – 27 (2003).

[11] Sandip Debnath, Prasenjit Mitra, Nirmal Pal, C. Lee Giles: Automatic Identification of Informative Sections of Web Pages. In: IEEE Transactions on Knowledge and Data Engineering, 17(9): 1233-1246 (2005).

[12] Shian-Hua Lin and Jan-Ming Ho: Discovering informative content blocks from web documents. In: Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining, pages 588–593 (2002).

[13] Suhit Gupta, Gail Kaiser, David Neistadt, Peter Grimm: DOM-based Content Extraction of HTML Documents. In: 12th International World Wide Web Conference, 12th International World Wide Web Conference (May 2003).

[14] Valter Crescenzi , Giansalvatore Mecca , Paolo Merialdo: RoadRunner: Towards Automatic Data Extraction from Large Web Sites. In: Proceedings of the 27th International Conference on Very Large Data Bases, p.109-118, September 11-14 (2001).

[15] World Wide Web Consortium. World Wide Web consortium hypertext markup language.

[16] Yanhong Zhai, and Bing Liu: Web Data Extraction Based on Partial Tree Alignment. In: Proc. The 14th international World Wide Web conference (WWW-2005), in Chiba, Japan10-14 (2005).

[17] Yves Weibig, Thomas Gottron: Combinations of Content Extraction Algorithms. In: Workshop Information Retrieval (2009).