Isomorphic mapping of DOM trees for Cluster-Based Page Segmentation

Jan Zeleny
Faculty of Information Technology
Brno University of Technology
Brno, Czech Republic
izeleny@fit.vutbr.cz

Radek Burget
Faculty of Information Technology
Brno University of Technology
Brno, Czech Republic
burgetr@fit.vutbr.cz

***Abstract*—In our previous work we have designed a method for fast and precise Web page segmentation. In this paper we propose a complementary algorithm and data structures that extend the original design. The extension is focused on isomorphic mapping between two DOM trees. Our main objective is to improve robustness of our original solution. We successfully design and implement a solution that is more robust while keeping the efficiency of the original simple one. To prove qualities of our new design we also offer an experimental evaluation of the new implementation.**

***Keywords*-vision-based page segmentation, cache, template detection, cluster-based page segmentation, DOM, tree mapping**

I. Introduction

In recent years, the quantity of information on the Web increased considerably. The problem is that the information on a Web page is rather unstructured and noisy which makes it difficult to perform tasks like information retrieval or content classification. There is much information on every Web page that can significantly distort machine understanding of the page.

Several algorithm types exist to deal with the problem, Template Detection and Page Segmentation being the most important two of them. Each of these algorithm types has its shortcomings. However using their right combination presents a solution with all these shortcomings minimized.

Vision-based page segmentation (VS) methods focus on splitting the Web page into coherent blocks. These can be then considered atomic information carriers. To properly do that, these algorithms try to comprehend the layout of the page in the same way human user would comprehend it when looking at the page. The main problem of these methods is scaling. They have to render each page[1], [2], [3] which is a very time-consuming task. Template detection[4], [5], [6] methods are much faster than VS but their granularity is often too coarse, as they just split the page in two parts and not individual segments. However some of these methods also offer a way to measure a "distance" from other pages to detect whether they are based on the same template.

When we look at the big picture, a set of pages is usually processed at once. In our previous work [7] we have designed an algorithm that crawls through the entire Web site, performs segmentation on some pages and stores the result of every segmentation it performs. When encountering other pages based on the same template, the algorithm just re-uses previously cached result instead of performing the segmentation again. To re-use the previous result, we need to isomorphically map Document Object Model (DOM) tree of the new page to DOM tree of the cached page. In our previous work we have used just a crude implementation of such mapping. No other references of mapping algorithms like the one we propose here have been found in existing literature, not even in papers that seem to be using it. In such cases authors probably use some intuitive approaches and therefore don't describe them explicitly.

In this paper we propose data structures and algorithms necessary to achieve the isomorphic mapping. The motivation of our work is to improve the original approach that is used in [7]. In this paper, we focus on space and time optimization of the mapping algorithm, as well as its robustness, since each page in the cluster may contain slight differences when compared to the segmented one.

Sections II and III cover concepts related to our work. To understand what are we trying to achieve, it is important to also understand these concepts. Sections IV and V describe the improved approach itself and finally section VI evaluates results of the proposed method.

II. Templates

The principle of templates is key for algorithms we propose. It is also very simple. Most of the modern Web sites are created using some sort of content management systems. A basic idea of such systems is that authors create content without any knowledge of the structure of Web pages on the site. Templates defined in the content management application itself are carriers of that structure. There is a number of templates for every site. Each template defines the core structure of the corresponding set of pages within that site. Some places in these structures are specially marked and a specific content is filled in them when a visitor displays any page based on the template.

The principle of templates directly supports the concept of consistent Web pages through the Web site. The consistency

of Web pages enhances user experience and orientation on the site[8]. That's why a vast majority of modern Web sites is based on templates[9].

III. Template caching and tree mapping

We use a processing model where the entire site is processed at once and the result of every performed segmentation is stored right after the segmentation is performed[7]. One cycle of this process is represented by the block schema displayed on figure 1. In this cycle we look if there is a "pattern page" in the cache which is based on the same template as currently inspected "input page". If there is no such pattern page, the input page is segmented, becomes one of the pattern pages and can be used in the following cycles. If there is one, no segmentation is performed and requested result is retrieved via isomorphic mapping between the input and the corresponding pattern.

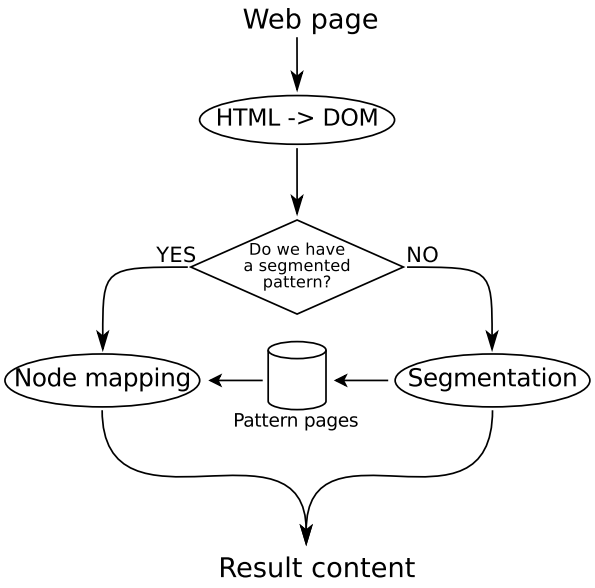

Figure 1. Block schema of site processing model

Considering that the algorithm already exists and we don't need to modify it, we focus only on the isomorphic mapping. On a closer look at the node mapping process itself, it is important to note that our approach doesn't perform generic tree-to-tree node mapping as described in the literature[10], [5]. We are trying to find a single node in given DOM tree with a generic DOM-tree-searching algorithm, having an address of requested node as an input. This simplification decreases time complexity of the mapping process and thus allows its performance increase.

The mapping process is displayed on figure 2 in slightly wider context. The left branch of the diagram corresponds to the left branch of the diagram displayed on figure 1, while the right branch loosely corresponds to the middle part of figure 1. Visual model refers to the output of utilized vision-based segmentation algorithm. The model is stored in the local cache together with corresponding DOM tree.

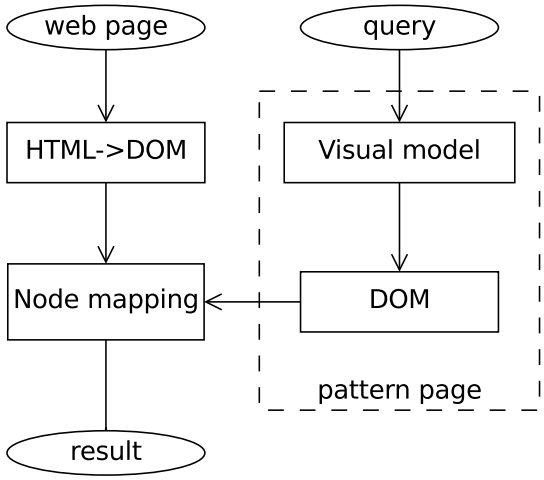

Figure 2. Block schema of tree matching

The data flow considered is as follows:

1) the processing algorithm receives input page and a pattern page (represented by a tree of visual areas) corresponding to it
2) a query executed by an external entity (user or process) selects a particular visual region of input page (e.g. "retrieve the heading of the article")
3) the DOM nodes associated with the visual region are retrieved and their addresses constructed
4) addresses from the previous step are used to locate the corresponding nodes in the input DOM tree
5) the content of located nodes from the input page is returned

Note that the main part (*node mapping* on figure 2) is represented by item four – the item number four is also the part we are focusing on. By mapping DOM nodes and extracting the content directly, we eliminate the need to segment the new page. Without this step the entire Cluster-based page segmentation[7] would not work.

IV. Node identification

Our design is strictly based on generic properties of DOM trees. This section analyzes various techniques which we use to find a particular node in a DOM tree. It is divided into three parts – first we cover location of the target node, then error detection mechanisms and the last part covers final verification.

There are two properties of DOM trees defined in the W3C recommendations[11], [12] which we can utilize. The first is that the order of sibling nodes has its significance and second that each DOM node has a label assigned. It usually corresponds to its HTML tag name. It is also important to

note that the DOM tree can contain other nodes besides those representing HTML elements, such as nodes representing element attributes. We need to drop all these to prevent malformation of the algorithm output.

Our design consists of two parts which have a strong connection with each other: *design of nodes' addresses* (further designated as descriptors) and *algorithms working with these descriptors*. There are several ways how to unambiguously identify node in a DOM tree. We look for a method (or combination of methods) that matches the following criteria:

- **speed:** localization of the node has to be as fast as possible
- **space:** the representation of the node location has to be space-saving
- **uniqueness:** results have to be unambiguous
- **reliability:** results have to be deterministic and error-prone

To elaborate on the last item: when we perform mapping between two pages $P1$ and $P2$ that are based on the same template, the method has to return either the same (corresponding) node or nothing – if there is a node on $P2$ which seems to correspond to a node on $P1$ according to the address but it is a different node from the perspective of user looking at the page, the method should detect such situation and return empty result.

Note that there is already an existing solution that is close to fulfill the requirement list. The XPath language addresses all the requirements. Taking that into account, we are looking for a solution that is even more robust and has a potential for a more time-effective implementation.

A. Path of positions

The path of positions is basically a string of numbers. Each number designates a position of a certain node within its siblings. The concept is based on the fact that every DOM tree is ordered. The first number designates the position of the node among children of the root node, the second one designates the position within child nodes of the one selected by the first number and so on.

This method meets almost all prerequisites for safe node identification. It offers great speed, its complexity is only $O(n)$ where n is a maximal depth of the tree. It is space saving since the representation of descriptor is just a series of numbers and it gives unique results due to the given order of siblings on each level. The only problem is limited guarantee that the returned result will be the correct one since we have no other clues to verify that the node is really the one we are seeking. We need some an detection mechanism to do that. That will be described in the following section.

B. Feature Fingerprints

Feature fingerprint can be defined as a combination of various attributes that DOM nodes can have. These attributes can be either visual (like size or position of rendered box corresponding to the DOM node), tree-based (e.g. tag name or value of any specific attribute) or content-based (e.g. word count). Specific combination of these features can be used to find node in a DOM tree or verify that we picked the right node. For example Hao et al. in [13] use combination of position within one level of the tree combined with visual position and size.

A good fingerprint has to be as unique as possible. The problem is that no fingerprint will be 100% reliable since there is no way how to identify a node with complete accuracy (the only exception would be if every node in the DOM tree had a unique id). Therefore we try to find such combination of properties that would maximize the probability of correct node identification while minimizing the number of attributes included in the fingerprint (the more attributes we add, the more space they take and the longer time their evaluation takes).

Besides the uniqueness we also want the fingerprint to be as generic as possible. Generic fingerprints are those that are applicable for as many nodes in the tree as possible, ideally all of them. Fingerprint is applicable to a node if the node contains some (ideally all) attributes that the fingerprint contains. There is also a third constraint for fingerprints – we ruled out all vision-based attributes like position on the page and size of the rendered box because getting their actual values would require rendering the page and therefore it would be too time-consuming. Even though ruling out vision-based attributes increases time efficiency, it may decrease the accuracy. In our experimental results we show that this problem doesn't appear in practical application. However if necessary, this decreased accuracy can be compensated by node verification described in section IV-C.

One feature is common to all nodes in the DOM tree and that is the name of corresponding HTML tag (p, strong, ...), let's call it element type. We use this element type in combination with value of `id` attribute (if the element in question has one) as a node fingerprint for each node on the path described in section IV-A.

C. Subtree Fingerprints

While generic feature fingerprints are somewhat reliable, there is a method that offers even bigger amount of reliability for most nodes in the DOM tree. It is a subtree-fingerprinting which is a very specific case of feature fingerprinting. Subtree fingerprinting is based on a simple premise:

Premise 1: Two nodes in two trees are equivalent if their position in their respective trees is at least approximately corresponding and if their content is equivalent.

This premise comes from the definition of templates – the core structure of two Web pages based on the same template is the same, therefore the position of all nodes belonging to the template will be the same on both pages. This rule doesn't apply for the content itself, as it is not part

of the core template structure. For the same reason it is not important to us, as segmentation algorithms are not supposed to select part of the content as a visual block. To deal with complex definition of content equivalence we came up with the following simplification.

Definition 1: Content of two nodes is considered equivalent if subtrees rooted at each of the two nodes are equivalent.

To evaluate this subtree equivalence, we use method of tree-path comparison used in [6]. It measures similarity of two DOM trees (or in this case their subtrees) by comparing their path sets. Each path in the set is a string of nodes starting at the root of the subtree and ending at one of the leaf nodes. This method was chosen for its superior performance over other methods used for the purpose of measuring the distance between the two DOM trees.

This verification method can help to significantly increase the accuracy of the mapping algorithm. However, the subtree fingerprinting is much more demanding than simple *type-id* feature fingerprinting, hence we use this only as a verification method for our results. It is important to note that this means we don't expect this step to be used in practical application of our algorithm unless someone requires even more strongly assured accuracy. When we perform the verification, we verify only the target node, not all nodes on the path leading to it.

V. Descriptors and algorithm

As we have outlined in section IV, our approach consists of two parts which are strongly connected to each other. This section will describe both of them.

A. Descriptors

The core of our descriptors is a position-based path. This representation is simple, it is basically string of numbers, as described in section IV-A. Such a string is formally represented as $P = (p_1, p_2, \ldots, p_n)$, where P is designation of the path. Its elements, designated p_X, are the numbers representing positions of their respective nodes. Implementation-wise, p_X is an index of a node in the array of child nodes on the Xth level of the DOM tree. Note that only child nodes of the previously selected node are in this array, not all nodes on that particular level of the DOM tree.

The shortcoming of this method (its inability to verify the located node) is addressed by feature fingerprinting. With the fingerprinting included in the model, the descriptor would contain the fingerprint of the last element in the path p_n. However in our implementation we go further and we extend the verification process to every single node along the path. Each node in the path is then represented by a structure $n = (p_X, id, type)$ where id contains the value of ID attribute and type contains name of the HTML element represented by the node.

On top of the described feature fingerprinting we add one more check and that is a total number of nodes on the particular level of DOM tree (i.e. $sibling\_count + 1$). This number is very useful indicator of a difference between the two trees. It is not 100% accurate but if the number is different from the actual count of nodes on that level of DOM tree, it always means that the tree is different than expected and the algorithm should fail.

Taking that into account, the final descriptor will be defined as follows:

Definition 2: The descriptor is a string of node-representing structures $P = (n_1, n_2, \ldots, n_n)$. Each node-representing structure is described as $n_X = (p_X, cnt_X, id_X, type_X)$ where cnt_X and p_X designate the total count of nodes within the sibling array and the position of node in this array respectively and id and $type$ represent feature-fingerprint of the node. X designates the depth of the node within the DOM tree.

B. Search algorithm

The search function is directly derived from the representation of Descriptors. Although all parts of the function were described in section V-A, it is due to write the function here as a whole. Algorithm 1 shows a formal declaration of the function written in pseudocode with syntax similar to Python.

Algorithm 1 Node search

```
function find_node(tree, node):
  cur = tree.root
  for each n in node.path:
    if length of cur.children > 0:
      return None
    cur = cur.children[pos]
    if cur.type != n.type or
       cur.id != n.id:
      return None

  if verify_node(cur, node) is not None:
    return cur
  else
    return None
```

The first half of the algorithm was already described and is therefore self-explaining. The second part implements verification of the result by the means of subtree-fingerprinting as described in section IV-C. Note that the algorithm returns empty result represented by the *None* value in case any of the verification checks fails.

C. Alternative search algorithm

As an alternative approach we devised a modified version of the search algorithm and Descriptor structure. This modified version (designated as *compact*) tries to optimize

space required for descriptors by trimming some of them. The algorithm trims the descriptor so it contains only the last node having non-empty id attribute and all the followings nodes. Using the id attribute is the only way to ensure the root node is correctly found.

The search algorithm in this case is the same as the one described in section V-B, the only difference is that the reference to the tree is replaced by a reference to the subtree rooted at the node found by depth-first search (DFS) algorithm:

Algorithm 2 Compact search – finding root of a descriptor

```
function compact_find(tree, node):
  subtree = dfs(tree, node.path[0].id)

  return find_node(subtree, node)
```

The *dfs* function on the first line of the function *compact_find* implements the depth first search. That is one of the few effective ways to find a node identified just by id in DOM tree without having any more information about its position. Our evaluation showed that breadth first search has similar time-consumption characteristics but DFS is slightly better.

VI. Experimental evaluation

We have implemented a program for experimental evaluation of our design. It is a part of our original implementation of Cluster-based page segmentation. It tests the previously described method for finding the node as well as various fingerprinting methods for error detection. The main part of the lookup uses the position-based path. All possible combinations of **id**, **type** and **count** are tested as fingerprinting error detection.

All the combinations are used in descending order of their precision, starting with the "full combination" of all three attributes, ending with the "empty combination" where none of them is used (i.e. no check is in fact performed). Note that more precise methods have higher risk of returning false-negatives (filtering out correct nodes). For each node in the path, the algorithm starts with the most restrictive combination. If it fails (i.e. it indicates an error) the less restrictive one is tried as a fall back. The same fall back mechanism is then used repeatedly until no error is detected. This approach allows us to track potential false-negatives. Besides this error detection for every node on the path we also use the final node verification via the subtree fingerprinting.

Several sites were chosen for testing (see table I for details). Most of them are news sites because the underlying segmentation mechanism has the best results with them. It is important to note that the choice of this segmentation mechanism doesn't influence quality of the mapping algorithm, it just influences the choice of testing sites. A set of 1500 pages was inspected on each site. We measured primarily the total number of mappings performed, and time consumption of each search. Since these statistics were measured on a per-template basis we had to put them together. The final number of matches is a sum of all template matches and the time is a weighted mean of time results of all templates. The table I also contains one other column and that is the number of nodes inspected by our error detection mechanisms. This number also illustrates the space saved by compact descriptors.

site	type	mappings	nodes	avg. time
iDnes.cz	full	5024	59499	7.6 ms
iDnes.cz	compact	5024	19375	0.64 s + 4.7 ms
novinky.cz	full	5130	40592	7.3 ms
novinky.cz	compact	5130	17741	1.1 s + 5.7 ms
theregister.co.uk	full	323	2044	30.7 ms
theregister.co.uk	compact	323	752	0.5 s + 20.1 ms
telegraph.co.uk	full	7140	66107	8.4 ms
telegraph.co.uk	compact	7140	31736	0.7 s + 4.1 ms
penize.cz	full	17537	177020	4 ms
penize.cz	compact	17537	80500	0.2 s + 2.2 ms

Table I
STATS OF THE NODE IDENTIFICATION ALGORITHM

Both full paths and alternative compact paths are included in the evaluation. For the compact method we separated time of root search from the path search. Both BFS and DFS were performed for each root node search and the lower time is displayed in the table. These results indicate that the lookup time of root paths makes these compact descriptors worse by two orders of magnitude.

Accuracy was evaluated as well. All hits were made by the most restrictive error detection: the combination of sibling count, HTML element name and (if possible) id match. That indicates no false-negatives are caused by the detection. After each target node was found, subtree fingerprinting was applied on it for verification of the result. The results were verified with this method in 100% of cases. A subset of tested pages (randomly selected 10% of the pages) were additionally verified by human, again with 100% success. That shows that the method doesn't produce false positives. Bottom line is that the method will very likely return the node corresponding to the one on the pattern page, if such node exists.

To sum up our results: full descriptors with the triple check based on *ID match*, *element name* and *sibling count* provides strong verification options while still being reasonably fast. That makes it the best approach of all considered alternatives.

VII. Conclusion

In this paper we analyzed several methods for finding nodes in DOM trees. These methods are evaluated and then combined in order to utilize all their strengths. In our evaluation we demonstrated that the final method is effective and can be used in practical applications.

Besides finding the desired node, this work is focused on error detection methods. The resulting error detection method is quite strong, as proven by our evaluation. By including error detection to the scope of this work we have laid down foundation for the next research which should be focused on methods for error correction.

Acknowledgement

This work was supported by the research programme MSM 0021630528, the BUT FIT grant FIT-S-11-2 and the IT4Innovations Centre of Excellence CZ.1.05/1.1.00/02.0070.

References

[1] W. Liu, X. Meng, and W. Meng, "ViDE: A vision-based approach for deep web data extraction," *IEEE Trans. on Knowl. and Data Eng.*, vol. 22, no. 3, pp. 447–460, March 2010.

[2] D. Cai, S. Yu, J. rong Wen, and W. ying Ma, "VIPS: a vision-based page segmentation algorithm," Microsoft technical report MSR-TR-2003-79, November 2003.

[3] R. Burget, "Layout based information extraction from HTML documents," in *Proceedings of the Ninth International Conference on Document Analysis and Recognition - Volume 02*, ser. ICDAR '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 624–628.

[4] L. Yi, B. Liu, and X. Li, "Eliminating noisy information in web pages for data mining," in *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, ser. KDD '03. New York, NY, USA: ACM, 2003, pp. 296–305.

[5] K. Vieira, A. S. da Silva, N. Pinto, E. S. de Moura, J. a. M. B. Cavalcanti, and J. Freire, "A fast and robust method for web page template detection and removal," in *Proceedings of the 15th ACM international conference on Information and knowledge management*, ser. CIKM '06. New York, NY, USA: ACM, 2006, pp. 258–267.

[6] T. Gottron, "Bridging the gap: from multi document template detection to single document content extraction," in *Proceedings of the IASTED International Conference on Internet and Multimedia Systems and Applications*, ser. EuroIMSA '08. Anaheim, CA, USA: ACTA Press, 2008, pp. 66–71.

[7] J. Zeleny and R. Burget, "Cluster-based page segmentation - a fast and precise method for web page pre-processing," in *Proceedings of the 3rd International Conference on Web Intelligence, Mining and Semantics*, ser. WIMS '13. New York, NY, USA: ACM, 2013.

[8] J. Nielsen, "User interface directions for the web," *Commun. ACM*, vol. 42, no. 1, pp. 65–72, Jan. 1999.

[9] D. Gibson, K. Punera, and A. Tomkins, "The volume and evolution of web page templates," in *Special interest tracks and posters of the 14th international conference on World Wide Web*, ser. WWW '05. New York, NY, USA: ACM, 2005, pp. 830–839.

[10] K. Vieira, A. L. Costa Carvalho, K. Berlt, E. S. Moura, A. S. Silva, and J. Freire, "On finding templates on web collections," *World Wide Web*, vol. 12, no. 2, pp. 171–211, June 2009.

[11] A. L. Hors, P. L. Hegaret, L. Wood, G. Nicol, J. Robie, M. Champion, and S. Byrne, "Document object model (DOM) level 3 document object model core," W3C Recommendation, April 2004.

[12] J. Stenback, P. L. Hegaret, and A. L. Hors, "Document object model (DOM) level 2 document object model html," W3C Recommendation, January 2003.

[13] Q. Hao, R. Cai, Y. Pang, and L. Zhang, "From one tree to a forest: a unified solution for structured web data extraction," in *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*, ser. SIGIR '11. New York, NY, USA: ACM, 2011, pp. 775–784.