A Scoring Map Algorithm for Automatically Detecting Structural Similarity of DOM Elements

Julián Grigera[1,2,3], Juan Cruz Gardey[1,2], Alejandra Garrido[1,2] and Gustavo Rossi[1,2]
[1]*LIFIA, Facultad de Informática, Universidad Nacional de La Plata, La Plata (CP 1900), Argentina*
[2]*CONICET, Argentina* [3]*CICPBA, Argentina*
{juliang, jcgardey, garrido, gustavo}@lifia.info.unlp.edu.ar

Keywords: Information Extraction, Web Adaptation, Refactoring for Usability

Abstract: Most documents in the WWW are generated from templates that represent user interface (UI) elements, and later filled with contents. In the field of information extraction, many approaches emerged to analyze the documents' structure, obtain similar features amongst them, and generate wrappers that are used to extract the raw contents from such documents. Therefore, most techniques documented in the literature are optimized to compare full documents, but there are other fields of applicability that require analyzing structural similarity on smaller UI components, like web augmentation or transcoding. In this paper we present two flexible algorithms to measure similarity between DOM Elements by using a mixed approach that considers both elements' location and inner structure. The proposed algorithms were used in the context of two projects: an approach for automatic usability refactoring, and a web accessibility helper. We also present a wrapper induction technique based on such algorithms. Additionally, we present a precision & recall evaluation of our algorithms as compared with other known approaches, applied to DOM elements of different sizes, but smaller than full scaled documents. The proposed algorithms run in linear time, so they are faster than most approaches that analyze structural similarity.

1 INTRODUCTION

The world changed profoundly when 25 years ago, Tim Berners-Lee publicized the World Wide Web project and invited wide collaboration (W3C, 2016). Thanks to that turning point in our history, today we possess the largest data repository for all people and all fields of science. However, being able to profit from this huge amount of data requires building increasingly complex automation techniques in two main areas of research that are of our interest: information extraction, and human-computer interaction (HCI) aspects like adaptability and usability.

In the area of information extraction, it is essential to recognize document structure and identify structural similarity, since this allows clustering functionally equivalent components and design cluster-specific mining algorithms for each cluster (Omer, Ruth, & Shahar, 2012; Joshi et al, 2003). For example, in a news portal that shows all articles in structurally similar components (in terms of inner elements and look-and-feel), this similarity could be identified by news extraction applications to separate presentation from content (Reis et al, 2004). Other approaches in the Area of Web Engineering even use variants of Web Scraping to make the application development process easier (Norrie et al, 2014). Meanwhile, in the field of HCI, there is an increasing demand to automatically improve the user experience in the web, including techniques for adaptability, personalization, accessibility, and usability. Among the research works in this area we may find techniques like *adaptation mechanisms* for touch-operated mobile devices (Nebeling, Speicher & Norrie, 2013), *transcoding* to improve accessibility (Asakawa & Takagi, 2008), *augmentation* to create personalized applications versions (Diaz, 2012) and our own work on *refactoring* to improve usability (Grigera et al 2016; Grigera, Garrido & Rivero, 2014) and accessibility (Garrido et al, 2013). For the sake of conciseness, in this article we will call all these techniques *web adaptations*. We claim that for these techniques to work correctly and fully unattended at a large scale, the ability to detect structural similarity in pages or smaller UI elements is essential. For example, if an e-commerce website has an accessibility problem in the UI element that displays

a product, it will inevitably have the same problem in all products, and it will be important to apply a solution to all instances, no matter the product. Therefore, in our research objective to automatically improve usability in web pages, we found that we could benefit from the work on data extraction, as we have a challenge in common: automatically and precisely discover the structure and similarity of web content to be able to cluster it before further processing.

It is interesting to note that web adaptations, even if they can be compared to "code adaptations" (i.e., refactorings and program transformations), have a fundamental difference from the approach used in code refactoring tools, in which code smells are detected and solved on a per-instance basis. Since UI elements from web documents are automatically generated from templates, as opposed to manually crafted code snippets, web adaptations at large should generally work like program restructuring transformations (Akers et al, 2005): all instances of a problem should be matched and fixed at once.

The challenge of detecting structural similarity is generally addressed by the fact that most data intensive websites generate their content dynamically, by retrieving it from a database and displaying it using template-generated pages. Thus, in the field of information extraction, several approaches were devised to discover the structure in the underlying templates. These approaches typically produce a *wrapper*, which is then used to match all structurally similar pages and extract their raw contents. Once the similar pages are grouped in clusters, the wrapper is generated by identifying the common structure within each cluster (Joshi et al, 2003). Several algorithms used to cluster similar pages compute tree edit distance between the DOM structure of entire pages (Omer, Ruth, & Shahar, 2012; Reis et al, 2004). Since these algorithms are computationally expensive, other approaches have been proposed to measure structural similarity with improved execution time, but mostly at the expense of accuracy (Joshi et al, 2003; Buttler, 2004).

Besides the problems with execution time, complexity of data structure and accuracy, most techniques for data extraction using structural clustering are effective when applied to full documents. Since the algorithms generally consider the inner structure of deep trees, they usually get poorer results in the comparison of smaller DOM elements that consist of only few nodes. However, web adaptation techniques like transcoding or refactoring are generally applied on such DOM elements. For example, the adaptations for mobile interfaces allowed by the tool W3Touch, may be used to resize all menu items with touch-related problems inside a page (Nebeling, Speicher & Norrie, 2013). Another example is the usability refactoring "Turn Attribute into Link" (Garrido, Rossi & Distante, 2011), which may be used to insert a link in all similar elements of a list.

The problem of comparing similarity between DOM elements has been addressed before, usually by comparing the elements' relative position in a document using XPath locators[1] (Grigalis & Čenys, 2014; Amagasa, Wen & Kitagawa, 2007; Zheng et al, 2009). However, while they may successfully detect similar elements arranged in iterative structures (like lists or menus), they are weaker in the cases where similar elements are placed in different locations. When this happens, the elements' inner structure can help to determine similarity where the location failed to do so, if the structure is complex enough.

In this paper we present an algorithm designed to detect similarity in DOM elements of different sizes. This algorithm is based on the comparison of both XPath locators and inner structure, including relevant tag attributes. Additionally, a variant of this algorithm is presented, which considers also on-screen dimensions and position of the elements. In order to show their applicability, we describe how we apply these algorithms in two approaches for web adaptation in usability and accessibility.

Our algorithms can successfully compare and cluster elements as small as single nodes but has also the flexibility to compare larger elements with the same accuracy, or even better than state-of-the-art methods. We support this claim with an evaluation and explain the results and performance differences with respect to other algorithms. Additionally, a wrapper induction technique is presented, that is based on the Scoring Map algorithm.

In summary, this paper we make the following contributions:

- We present two algorithms that compute a similarity measure between DOM elements and perform well with different element sizes.
- We show the applicability of the algorithms in two different projects for web usability and accessibility. The DOM element comparison is used for clustering elements with the same usability smell in one case, and accessibility

[1] XML Path Language http://www.w3.org/TR/xpath-31/ (accessed Aug 25, 2016)

smell in the other; the inducted wrapper is then used to fix the bad smells by refactoring or transcoding.
- We assess the performance of the algorithms in comparison to others found in the literature through an empirical evaluation, with a set of more than 800 DOM elements of different sizes from well-known sites.

2 RELATED WORK

Existing literature in the area shows several different ways of detecting similarity between DOM elements. Some of these techniques are applicable to any tree structure, and some are specific to XML or HTML structures. Many of these methods can be found in an early review by Buttler (2004). We present the related work in three groups: tree edit distance algorithms, bag of paths methods, and other approaches.

2.1 Tree Edit Distance Algorithms

Since DOM elements can be represented as tree structures, they can be compared with some similarity measure between trees, such as edit distance. This technique is a generalization of string edit distance algorithms like Levenshtein's (1966), in which two strings are similar depending on the amount of edit operations required to go from one to the other. Operations typically consist in adding or removing a character and replacing one character for another.

Since the Tree Edit Distance algorithms are generally very time-consuming (up to quadratic time complexity), different approaches emerged to improve their performance. One of the first or such algorithms proposed by Tai (1979) uses *mappings*, i.e. sets of edit operations without a specific order, to calculate tree-to-tree edition cost. Following Tai's, other algorithms based on mappings were proposed, mainly focused on improving the running times. A popular one is RTDM (Restricted Top-Down Distance Metric) (Reis, 2004), which uses mappings such as Tai's but with restrictions on the mappings that make it faster. Building on RTDM, Barkol, Bergman & Shahar (2012) developed SiSTeR, which is an adaptation that weights the repetition of elements in a special manner, in order to consider as similar two HTML structures that differ only in the amount of similar children at any given level (e.g. two blog posts with different amounts of comments).

Other restricted mapping methods were developed, like the bottom-up distance (Valiente, 2001), but RTDM and its variants are more suited for DOM elements where nodes closer to the root are generally more relevant than the leaves.

Griasev and Ramanauskaite modified the TED algorithm to compare HTML blocks [Griasev18]. They weight the cost of the edit operations depending on the tree level in which they are performed, giving a greater cost to those that occur at a higher level. Moreover, since different HTML tags can be used to achieve the same result, their version of the algorithm also accounts for tag interchangeability.

TED algorithms have also been proposed to compare entire web pages [Xu17]. This work defines the similarity of two web pages as the edit distance between their block trees; structures that contain both structural and visual information.

2.2 Bag of Paths

Another approach to calculate similarity between trees is by gathering all paths / sequences of nodes that result from traversing the tree from the root to each leaf node, and then comparing similarity between the bags of paths of different trees. An implementation of this algorithm for general trees was published by Joshi et al (2003), with a variant for the specific case of XPaths in HTML documents. The latter was used in different approaches for documents clustering (Grigalis & Čenys, 2014).

The *bag of paths* method has one peculiarity: the paths of nodes for a given tree ignore the siblings' relationships, and only preserve the parent-child links. This results in a simpler algorithm that can still get very good results in the comparisons. We have similar structural restrictions in our algorithms, as we explain later on in Section 3. The time complexity of this method is $O(n^2N)$, where n denotes the number of documents and N the number of paths.

Another proposal similar to the Bag of Paths approach in the broader context of hierarchical data structures, is by using *pq-grams* (Augsten, Böhlen & Gamper, 2005). Pq-grams are structurally rich subtrees that can be represented as sequences. Each tree to be compared is represented as a set of pq-grams, then used in the comparison. This performs in $O(n \log n)$, where n is the number of tree nodes.

Our approach is similar to the aforementioned in how it generates linear sequences to represent and eventually compare tree structures, although relying less on topological information and more on nodes' specific information (such as HTML attributes).

2.3 Other Approaches

Different approaches have been developed that have little or nothing to do with tree edition distance or paths comparison. Many approaches like the one

proposed by Zen et. al [Zen2013], rely on visual cues from browser renderings to identify similar visual patterns on webpages without depending on the DOM structure. Another interesting work is that of using fingerprinting to represent documents (Hachenberg & Gottron, 2013), enabling a fast way of comparing documents without the need of going through them completely, but by comparing their hashes instead. Locality preserving hashes are particularly appealing in this context, since the hash codes change according to their contents, instead of avoiding collisions like regular hash functions do.

Most of the previous methods focus on the inner structure of documents, but few consider external factors (amongst the ones commented here, only the work of Grigalis and Çenys (2014) use inbound links to determine similarity). This is due to the fact that these approaches are generally designed to compare full documents. There are however some works focused on comparing smaller elements where external factors like location play a more important role. Amagasa, Wen and Kitagawa (2007) use XPath expressions to identify elements in XML documents, which is an effective approach given that XML definitions usually have more diverse and meaningful labels than, for instance, HTML. Other works focused on HTML elements also use tag paths: Zheng, Song, Wen, and Giles generate wrappers for small elements, to extract information from single entities (Zheng et al, 2009). In this work the authors propose a mixed approach based on a "broom" structure, where tag paths are used identify potentially similar elements, and then inner structure analysis is performed to generate wrappers.

Our approach is also a combined method that considers some of the inner structure of the elements, but also their location inside the document where they are contained.

3 A SCORING MAP ALGRITHM TO DETECT DOM ELEMENTS

The proposed algorithm for detecting similarity between DOM elements is a weighted comparison of two aspects: location inside the DOM tree, and inner structure. As the inner structure grows larger, the more relevant it becomes in the final score. Conversely, when the structure is smaller, the location path that gains more relevance. This adjustable scoring method makes the algorithm flexible, enabling it to detect both small and large DOM elements in terms of tree structure. In this section we also introduce a variant that considers also the elements' dimensions and position as rendered in the screen. Finally, we show a wrapper induction technique based on the proposed algorithm and analyze the time complexity.

3.1 Scoring Map Algorithm

The proposed algorithm processes the comparison in two steps: first, it generates a map of the DOM elements to be compared, where some fundamental aspects are captured, such as tag names organized by level (i.e., depth within the tree), along with relevant attributes (e.g. *class*), but ignoring text nodes. Then, these maps are compared to each other, generating a similarity score, which is a number between 0 and 1. The main benefit of constructing such map structure, which is created in linear time, is that it enables computing the similarity measure also in linear time (with respect to the number of nodes).

The map summarizes key aspects of the elements' structure and location. Considering a single DOM element's tree structure, the map captures the following information for each node:

- **Level number**, which indicates depth in the DOM tree, where the target node's level is 0, its children 1, and so on. Parent nodes are also included using negative levels, i.e. the closest ancestor has level -1.
- **Tag name**, often used as label in general tree algorithms.
- Relevant **attributes**, in particular CSS class.
- **Score**, which is a number assigned by the algorithm representing the relevance of the previous attributes when comparison is made.

In the map, the **level number**, **tag name** and **attributes** together compose the **key**, and the **score** is the **value**, but since HTML nodes can contain many attributes, there will be one key for each, with the same level number and tag name. This way. a node will in fact generate many entries in the map, one for each attribute. For example, if a DOM element contains the following node:

```
<div id="container" class="main zen">
```

the map is populated with 3 entries: one for the `div` label alone, one for the `div` label with the `id` attribute, and 2 more for the `class` attribute, one for each value: `main` and `zen`. Only values of the `class` attribute are considered, other attributes are kept without their values. In this case, all three entries would get a same score (later in this section we explain how this is determined). Also, if this element were repeated, only one set of entries would be entered since a map cannot have repeated keys, which is actually a desirable property for an algorithm that applies to HTML elements. Documents generated by HTML templates typically contain iteratively generated data, e.g., comments in a post. In this case,

two posts from a same template should always be considered similar, no matter the different amounts of comments on each one. Therefore, comparing only one entry for a set of equivalent DOM elements is likely to obtain better results than considering them as distinct (Omer, Ruth & Shahar, 2012).

This map organization is important in the second step of the algorithm, where the actual comparison is made based on these scores. A full example of a DOM element and its scoring map is depicted in Fig. 1.

Two initial score values are set in the map: one at the root, and one at the first parent node (levels 0 and -1, respectively). These scores decrease sideways, i.e. from the root down to the leaves, and from the first parent node up to the higher ancestors. An important aspect at this stage is how the initial score for the parent node is generated, which is inversely proportional to the height of the tree. This is key to give our algorithm the flexibility to detect similar elements relying less on their location when the trees are large enough. In these cases, the inner structure scores get more weight on the overall comparison.

Another aspect worth mentioning of the map structure for the elements is how the tree structure is captured. Notice that the only information for each node regarding this structure is the level number (or depth), which ignores the parent-child relationships and also the order. The preliminary tests we ran, and finally the experiment described in the following section, showed that this makes the algorithm simpler and faster, and does not implicate a significant decrease in the obtained scores.

The final similarity score between the two elements is made by comparing the values of their maps. The following formula describes how we obtain the similarity **S** between two elements once their maps **m** and **n** are generated:

$$S(m,n) = \frac{\sum_{k \in (K(m) \cap K(n))} \max(m[k], n[k]) * 2}{\sum_{k \in K(m)} m[k] + \sum_{k \in K(n)} n[k]}$$

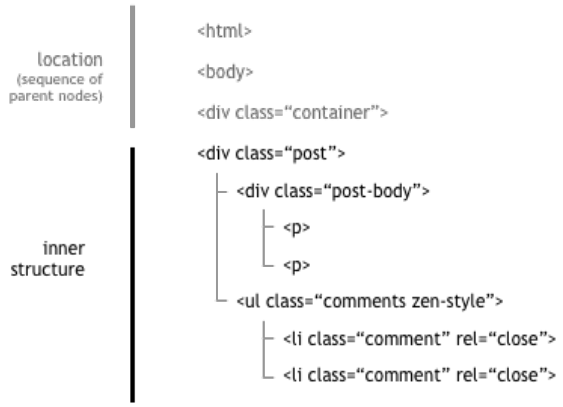
The function *K(m)* answers the set of keys of map *m*, and *m[k]* returns the score for the key *k* in the map *m*. In the dividend summation, we obtain the intersection of the keys for both maps. For each of these keys, we obtain the scores in both maps and get the highest value (with *max(m[k],n[k])*) times 2. The divisor term adds the total scores of both maps. This function intends to compare similitude in maps using their common keys over the number of combined keys, much like the Jaccard index calculates the ratio between intersection and union in sample sets. It is also to the way the Bag of Paths algorithm calculates similarity (Joshi et al, 2003).

3.2 Dimensional Variant

We devised an alternative algorithm that also considers size and position of the elements to improve the detection of similar elements when their inner structure is scarce. This criterion considers on-screen position and dimension to determine if two elements are part of a series of repetitive, similar widgets.

It is usual for repeating DOM elements to be aligned, either vertically or horizontally. In such cases, it is also usual for one of their dimensions to be also equal; in the case of horizontal alignment, the height (e.g. a top navigation menu), and in the case of vertical alignment, the width (e.g. products listing).

By using these properties in repetitive elements, we obtain a dimensional similarity measure between 0 and 1 for either of the two cases (i.e. vertical or horizontal alignment), in the compared elements.



Key	Value
-2@body	35.0
-1@div.container	70.0
-1@div	70.0
0@div	30.0
0@div.post	30.0
1@div	21.0
1@div.post-body	21.0
2@p	14.7
2@p	14.7
1@ul.comments	21.0
1@ul.zen-style	21.0
2@li	14.7
2@li.comment	14.7
2@li-rel	14.7

Figure 1. Sample DOM Element and its corresponding comparison map.

For each of the two potential alignments, this measure is calculated by obtaining the absolute values of the proportional differences in position and dimension. The higher of both alignment measures is then considered as an extra weighted term to the similarity formula. A visual example is shown in Fig. 2.

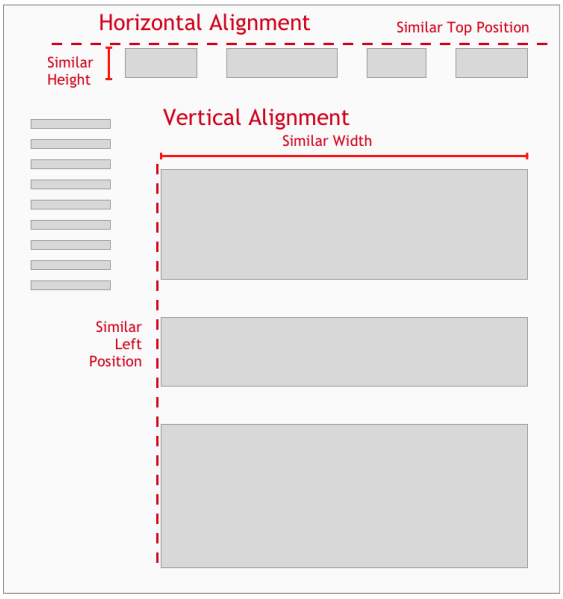


Figure 2. Dimensional alignment example in DOM Elements.

In current web applications, it is usual to find responsive layouts that adapt their contents to different devices. This could harm the dimensional scores when different sized elements are the same but rendered in different devices. Depending on the case, this may lead to wrong results, but also can be beneficial to detect them as different elements, e.g. in the field of applicability of web adaptation where different adaptations can be applied to the UI depending on the device.

This variant has shown improvements over the base algorithm in elements with little structure, but imposes an extra step in the capture to gather the elements' bound boxes. The base algorithm, on the other hand, can be applied to any DOM element represented with the HTML code only.

3.3 Time Complexity

The algorithm runs in O(n), i.e. linear time, with respect to the size of the compared trees (being n the total number of nodes). Even if there are two steps involved, one for building the maps and a second to perform the comparison, both are linear. It should be noticed that calculating the intersection of keys for the second step (function *K(m)* in the formula) is linear given that only one of the structures is traversed, while the other is accessed by key, which is generally considered constant (*O(1)*) for maps or dictionaries (although it can be, depending on the language, linear with respect to the size of the key).

When applying the algorithm to larger DOM structures, e.g. full documents, some improvements could be made to make this time worst case O(n) in practice, by applying a cutoff value when the relevance score drops below a certain level. This could reduce the trees traversing significantly, and consequently decrease the size of the maps, while keeping the most relevant score components.

3.4 Wrapper Induction

Using the algorithms described in this section, we developed a way of generating wrappers that may be later used to match new elements or retrieve raw information from them.

Initially, a first element is taken as reference to generate an initial map (the same way we explain in section 3.1) that will be used as *base map*. Then, as new elements are taken into consideration, we refine the base map to iteratively generate the wrapper. The way to achieve this is by generating a new map for each new element (we will refer to as *candidate map*) and apply the comparison algorithm, also described in section 3.1. Depending on the result of this comparison, we apply either positive or negative reinforcements over the base map.

When comparing a candidate map with the base map, if the result of the comparison is over a given similarity threshold, we will first find all the intersecting keys of both maps. The scores for these keys on the base maps are positively reinforced, by adding a reinforcement value. This value is calculated as a proportion of the score for the same key in the candidate map. During our experiments, we have obtained best results with a proportion of 0.35, but further analysis should be made to assess and optimize this value. When the result of the comparison between candidate and base maps do not reach the similarity threshold, a negative reinforcement is applied in a similar way, also to the intersecting keys, by subtracting the same reinforcement value. It is important that, no matter how much negative reinforcement a score gets, it never reaches a value below zero.

Intuitively, those keys that are shared among similar elements, will get a higher score once the wrapper induction is done. Conversely, those keys that are too common, i.e. present in many different elements in the document, will get a lower score, until eventually reaching zero. This way, only the distinctive keys (which represent tags and attributes) end up being the most relevant in the wrapper.

4 APPLICABILITY

We developed and refined our algorithms to measure structural similarity in the context of two different Web Engineering approaches that make use of web adaptation in the specific areas of usability and accessibility. These approaches are described in the next subsections.

4.1 Automatic Detection and Correction of Usability Smells

The first use case for the Scoring Map algorithm is automatic detection of usability problems on web applications. This was developed to ease usability assessment of web applications, since it is usually expensive and tedious (Fernandez, Insfran & Abrahão, 2011). Even when there are tools that analyze user interaction (UI) events and provide sophisticated visualizations, the repair process mostly requires a usability expert to interpret testing data, discover the underlying problems behind the visualizations, and manually craft the solutions.

The approach is based on refactoring and its capacity to incrementally improve not only internal quality factors of a deployed application, but also external ones, like usability (Distante et al, 2014) or accessibility (Garrido et al, 2013). We build on the concept of “bad smell” from the refactoring jargon and characterize usability problems as “usability smells”, i.e., signs of poor design in usability with known solutions in terms of usability refactorings (Garrido, Rossi & Distante, 2011).

The scoring map algorithm is then used in the tool that supports this approach, the Usability Smell Finder (USF). This tool analyses interaction events from real users on-the-fly, discovers usability smells and reports them together with a concrete solution in terms of a usability refactoring (Grigera, Garrido & Rivero, 2014). For example, USF reports the smell "Unresponsive Element" when an interface element is usually clicked by many users but does not trigger any actions. This happens when such elements give a hint because of their appearance. Typical elements where we have found this smell include products list photos, website headings, and checkbox/radio button labels.

Each time USF finds an instance of this smell in a DOM element, it calculates the similarity of this element with clusters of elements previously found with the same smell. When the number of users that run into this smell reaches certain threshold, USF reports it suggesting the refactoring "Turn Attribute into Link".

The wrapper induction technique presented in this paper becomes useful at a later stage. The toolkit is able in some cases to automatically correct a reported usability smell by means of a client-side web refactoring (CSWR) (Garrido et al, 2013), i.e., generic scripts that are parameterized to be applied on DOM elements in the client-side. Our proposal includes applying CSWR automatically by parameterizing them with the specific details of a detected usability smell and making use of the wrapper to find all matching elements that suffer from a same specific smell.

We obtained better results with the Scoring Map algorithm than simple XPath comparison, given that it works better on large elements, resisting HTML changes (small elements get similar results, since Scoring Map works in a similar way in these cases).

4.2 Use of Semantic Tags to Improve Web Application Accessibility

According to W3C accessibility standards, most Web applications are neither accessible nor usable for people with disabilities. One of the problems is that the content of a web page is usually fragmented and organized in visually recognizable groups, which added to our previous knowledge, allows sighted users to identify their role: a menu, an advertisement, a shopping cart, etc. On the contrary, unsighted users don't have access to this visual information. We developed a toolkit that includes a crowdsourcing platform for volunteer users to add extra semantic information to the DOM elements (Zanotti, 2016) using predefined tags, in order to transcode the visual information into a more accessible data presentation (plain text). This extra information is then automatically added on the client-side on demand, which is aimed to improve some accessibility aspects such as screen-reader functionalities.

By using an editor tool, when a user recognizes a DOM element, the tool applies the structural similarity algorithm to find similar elements that should be identically tagged within the page. For example, as soon as the user tags a Facebook post, all other posts are recognized as such and highlighted with the same color. At this stage we applied our wrapper induction method on the selected element to automatically tag the similar ones. Small structural differences are ignored, like the number comments. By adjusting the threshold, users can cluster elements with higher precision, e.g., successfully filtering out ads disguised as content. The Scoring Map algorithm was actually first developed during this work, since other comparison methods didn’t have an adjustable similarity threshold, or the required speed.

A screenshot of the tool capturing YouTube videos thumbnails can be seen in Fig. 3, where the threshold is adapted to include or exclude video lists in the clustering.

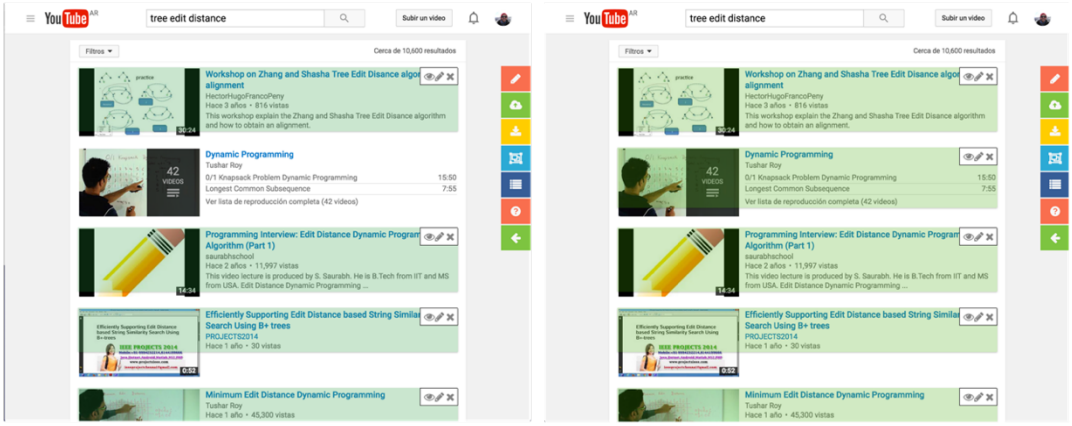


Figure 3. Web Accessibility Transcoder capturing similar DOM elements, using two different threshold values.

This way, we can easily reduce the entire web page into a limited set of semantically identifiable UI elements that can be used to create accessible and personalized views of the same web application, by applying on-the-fly transformations to each semantic element of the requested page. This happens in real time, as soon as the page loads, even on DOM changes, such as continuous content loading applications like Facebook or Twitter. On each refresh, the application applies different strategies to look for DOM elements that match the previously defined semantic elements, by applying the induction wrapper, the application is able to recognize those similar DOM elements where the new semantic information has to be injected. This process must be fast enough to avoid interfering on the user experience, which is achieved with our method.

5 EVALUATION

We ran an evaluation to compare our Scoring Map approach with a set of baseline algorithms. Since we did not find any data set with small UI elements in web templates, we created a new one. The evaluation compared how each algorithm clustered the elements with respect to a reference clustering.

5.1 Preparation

The data set was generated by capturing several DOM elements from different popular websites taken from the Alexa's top sites ranking[2]. To carry out this task, we created an assistance tool that allowed highlighting, capturing, and grouping DOM elements from any website. Using this method, we generated 124 groups with a total 818 DOM elements. We deliberately created groups with elements of different heights, considering the longest path amongst all the elements' DOM trees in the group. Grouped elements were kept alongside the set of all the elements ungrouped and shuffled to test the automatic grouping of the different algorithms during the experiment. Table 1 shows details of the dataset.

Table 1. DOM Elements' data set.

Elements' Height	*Groups Count*	*Elements Count*
1	44	334
2	28	201
3	15	76
4	13	65
5	12	60
6	7	39
7	4	40
Totals	***124***	***818***

Besides creating the dataset, we implemented the baseline algorithms to which we compared ours. The selected algorithms were RTDM (Reis, 2004), Bag of XPaths (Joshi et al, 2003) and simple XPath comparison. The latter was included for the sake of comparing our proposal to at least one other algorithm that was aware of the elements' location inside the DOM tree. Using this algorithm, two DOM elements are considered similar if their XPaths match, considering only the labels, i.e., ignoring the indices. For example, the following XPaths:

[2] Alexa's top 500 sites on the web http://www.alexa.com/topsites (accessed Apr, 2021)

```
/body/div[1]/ul[2]/li[3]
```

```
/body/div[2]/ul[2]/li[2]
```

would match when using this criterion, since only the concrete indices differ.

Except for the XPaths comparison algorithm, all other algorithms required setting a similarity threshold that was a number between 0 and 1. We ran several trials with different configurations to adjust these thresholds. For the RTDM strategy, we obtained the best results using a 0.8 threshold, while for the Bag of XPaths strategy was 0.6. In our algorithms the optimum threshold was higher: 0.90 for the Scoring Map algorithm and 0.85 for the Dimensional variant. Regarding the Dimensional variant, the weight assigned to the dimensional similarity measure was 0.2, leaving 0.8 for the similarity measure of the base algorithm.

For the sake of reproducibility, we have made both the data set and the source code of all implementations used in this experiment available online[3], including the capturing tool to expand the data set.

5.2 Procedure

We ran all 5 algorithms (the 3 baseline ones, plus the 2 proposed in this paper) on the dataset, and then analyzed the clustering that each one produced. The clustering procedure consisted in adding the elements one by one; if a group was found in which at least one element was considered similar, then the new element was added to that group, otherwise, a new group with the new element was created. Since for some algorithms the clustering is prone to change depending on the order in which the elements are added, we ran these algorithms with 30 different random orders and obtained the average numbers, and also calculated Standard Deviation to find the degree of this alteration in the results. The results for the XPaths algorithm showed no alteration whatsoever with different orders of elements, given that in this case there is no similarity metric involved but equality comparisons.

The way of comparing the automatically generated clusterings with the reference one was by an analysis of precision and recall, averaged by a f-score. Since this experiment did not qualify as a simple classification problem with predefined classes, there is a special interpretation for the terms that require clarification. The precision and recall analysis was calculated considering pairs of elements: given two elements, if both the automated and reference clustering put them in the same group, then we consider this case a **true positive**. A **true negative** is found when the opposite happens. If both elements are in the same group in the reference clustering but not in the automated clustering, we consider it a **false negative**. Finally, when the opposite happens, i.e., the automated clustering incorrectly groups the two elements together when the reference clustering does not, we consider this case a **false positive**. Thanks to this method, we were able to calculate precision, recall and f-score for the automated clustering methods.

Given the interpretation for false/true positives and negatives, the formula for calculating the precision and recall metrics is standard. In the case of the f-score, we specifically use a F1-Score formula, since we want to value precision and recall the same:

$$F_1 = 2 * \frac{precision * recall}{precision + recall}$$

This value represents a weighted average of precision and recall measures.

In addition to the evaluations over the complete set of elements, we ran additional tests with the sub-groups of elements by height, to gather specific metrics to analyze the potential influence of the elements' size on the different algorithms.

5.3 Results

For all the algorithms we measured precision, recall and f-score. Additionally, we captured the amounts of false positives and false negatives. The results for each algorithm can be seen in both Table 2 and Fig. 4, sorted by ascending f-score. It is important to remark that, as we explained earlier, these values are **averaged** from a set of 30 executions with different elements' order (except for the XPath algorithm), so obtaining an f-score by applying the formula to the Precision and Recall values on the corresponding columns will not exactly match the values on the f-score columns.

There is a very pronounced difference between the algorithms that consider the elements' inner structure, more prepared for comparing full documents, and the algorithms that consider the position of the elements, namely XPath, Scoring Map and Scoring Map variant with dimensional comparison.

[3] Experiment's resources available at: https://bit.ly/scoring_map

Table 2. Experiment's results.

Strategy	*Precision*	*Recall*	*F-Score*	*False Positives*	*False Negatives*
Bag Of XPaths	0.2334	0.9332	0.3735	11279	245
RTDM	0.2401	0.9225	0.3810	10750	285
XPaths	0.8411	0.9101	0.8742	633	331
Scoring Map	1.0000	0.9169	0.9567	0	305
Scoring Map (Dimensional)	0.9971	0.9265	0.9605	9	270

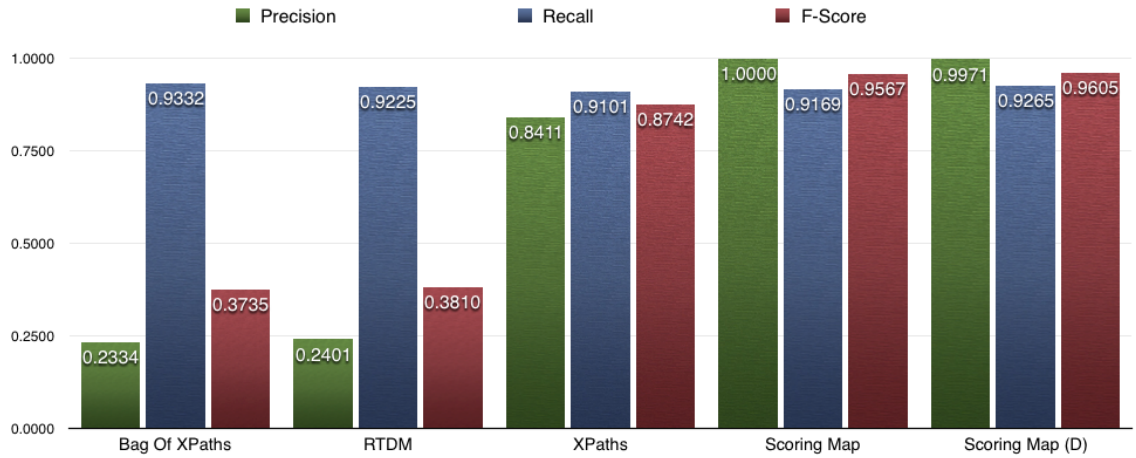


Figure 4. Precision, Recall and F1 Score of the algorithms.

Both algorithms presented in this paper outperformed the baseline ones in terms of precision and f-score, but not in recall. Notice though that high precision means lower false positives ratio, which in the context of this experiment indicates that fewer elements considered to be different (in the reference grouping) were grouped together by the algorithm. High recall, on the other hand, means that most matching couples of elements were correctly grouped together by the algorithm, even if it means that many other elements were incorrectly grouped together. A very sensitive algorithm, i.e. one likely to consider any couple of elements similar, will get this kind of results.

The dimensional variant of the Scoring Map algorithm reached the highest f-score (0.9605), but only slightly higher than the original algorithm. It also obtained a higher recall (0.9265), only second to Bag of XPaths considering all algorithms. The scoring map algorithm was however very close in f-score (0.9567), and obtained perfect precision. The worst performing algorithms in this analysis also showed high recall values, but when precision drops to low levels (below 0.3) this indicates a very high false positives ratio, as shown in Table 2.

Regarding the evaluation by level, we observed that, as expected, the baseline algorithms perform poorly when height is 1 (single nodes), but quickly ascends as height increases. Our algorithms keep a high score at all levels, showing their flexibility. The results by level are shown in Fig. 5.

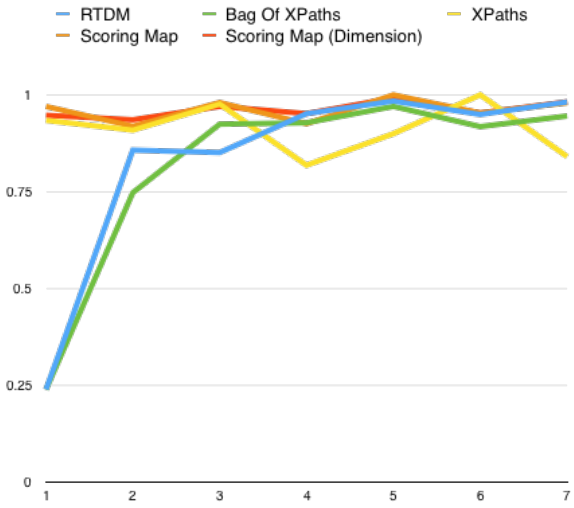


Figure 5. F-Scores by DOM elements' height.

With respect to the averaged results, we analyzed the Standard Deviation to assess the variations in the results with different orders of elements, but we found that it to be very low in all cases. The standard deviation in the f-score for the resulting clusters depending on the order was 0.0009 for both RTDM and bag of XPaths, 0.0016 for our scoring map algorithm and 0.0028 for the dimensional variant.

Further experiments with full documents should be performed to assess the adequacy of the structural algorithm (although the dimensional variant would make no difference in this case) where the baseline algorithms get the best results, according to the literature.

5.4 Threats to Validity

When designing the experiment, we faced many potential threats that required special attention. Regarding the construction of the elements' set, we had to make sure the reference groups were not biased by interpretation. To reduce this threat, at least two authors acted as referees to determine elements' equivalence in the reference groups.

Another potential threat is the size of the dataset itself. Since there is manual intervention to create the groups of elements, and there were also restrictions with respect of their heights, building a large repository is very time-consuming. The set is large enough so adding new elements does not alter the results noticeably, but a larger set would prove more reliable.

There is also a potential bias due to the clustering algorithm, which is affected by the order in which elements are supplied to the algorithms. We tacked this by running the algorithms repeatedly, as explained in section 5.2.

6 CONCLUSIONS

In this paper we presented an algorithm to compare DOM elements by similarity, especially optimized for different sized elements, flexible enough to work in small and larger ones. We have also presented a variant that improves for some cases the results and shown a wrapper induction technique based on these algorithms. Their implementations are simpler than most Tree Edit Distance algorithms and run in order O(n), being n the total number of nodes of both trees added together.

We have shown how we benefit from this algorithm in two projects that represent different scenarios in the web realm: automatic usability evaluation, and accessibility improvement by transcoding techniques.

By running an experiment with a set of 818 DOM elements we evaluated the performance of the proposed algorithm in comparison with 3 other known approaches. We used a data set to test the algorithms' flexibility including single DOM elements from well-known websites. The results show that our approach noticeably outperforms the baseline algorithms for small DOM elements and keeps high scores even as the elements grow up to 7 degrees high.

Thanks to the simple map comparison technique, the algorithm is relatively easy to implement, especially in contrast to the known tree edit distance approaches. It is also very flexible since it allows to weight the two comparison aspects.

In the future, we plan to assess the adequacy of the proposed algorithms to larger elements, in particular full documents. Since the algorithm is flexible and easy to extend, for instance to recognize topologies in a stricter way (i.e. incorporating parent-child relationships), control the weighted scores, etc. we think this could be a valid alternative for this and other unexplored scenarios.

ACKNOWLEDGEMENTS

The authors acknowledge the support from the Argentinian National Agency for Scientific and Technical Promotion (ANPCyT), grant number PICT-2019-02485.

REFERENCES

Akers, R. L., Baxter, I. D., Mehlich, M., Ellis, B. J., & Luecke, K. R. (2005, November). Re-engineering C++ component models via automatic program transformation. In *12th Working Conference on Reverse Engineering (WCRE'05)* (pp. 10-pp). IEEE.

Amagasa, T., Wen, L., & Kitagawa, H. (2007, September). Proximity search of XML data using ontology and XPath edit similarity. In *International Conference on Database and Expert Systems Applications* (pp. 298-307). Springer, Berlin, Heidelberg.

Asakawa, C., & Takagi, H. (2008). Web Accessibility: A Foundation for Research, chapter Transcoding.

Augsten, N., Böhlen, M., & Gamper, J. (2005, August). Approximate matching of hierarchical data using pq-grams. In *Proceedings of the 31st international conference on Very large data bases* (pp. 301-312).

Burzacca, P., & Paternò, F. (2013, July). Remote usability evaluation of mobile web applications. In *International Conference on Human-Computer*

Interaction (pp. 241-248). Springer, Berlin, Heidelberg.

Buttler, D. (2004, June). A short survey of document structure similarity algorithms. In *International conference on internet computing* (Vol. 7).

Díaz, O. (2012, July). Understanding web augmentation. In *International conference on web engineering* (pp. 79-80). Springer, Berlin, Heidelberg.

Distante, D., Garrido, A., Camelier-Carvajal, J., Giandini, R., & Rossi, G. (2014). Business processes refactoring to improve usability in E-commerce applications. *Electronic Commerce Research*, *14*(4), 497-529.

Fernandez, A., Insfran, E., & Abrahão, S. (2011). Usability evaluation methods for the web: A systematic mapping study. *Information and software Technology*, *53*(8), 789-817.

Garrido, A., Rossi, G., & Distante, D. (2010). Refactoring for usability in web applications. *IEEE software*, *28*(3), 60-67.

Garrido, A., Firmenich, S., Rossi, G., Grigera, J., Medina-Medina, N., & Harari, I. (2012). Personalized web accessibility using client-side refactoring. *IEEE Internet Computing*, *17*(4), 58-66.

Griazev, K., & Ramanauskaitė, S. (2018, November). HTML Block Similarity Estimation. In *2018 IEEE 6th Workshop on Advances in Information, Electronic and Electrical Engineering (AIEEE)* (pp. 1-4). IEEE.

Grigalis, T., & Čenys, A. (2014). Using XPaths of inbound links to cluster template-generated web pages. *Computer Science and Information Systems*, *11*(1), 111-131.

Grigera, J., Garrido, A., Panach, J. I., Distante, D., & Rossi, G. (2016). Assessing refactorings for usability in e-commerce applications. *Empirical Software Engineering*, *21*(3), 1224-1271.

Grigera, J., Garrido, A., & Rivero, J. M. (2014, July). A tool for detecting bad usability smells in an automatic way. In *International Conference on Web Engineering* (pp. 490-493). Springer, Cham.

Hachenberg, C., & Gottron, T. (2013, October). Locality sensitive hashing for scalable structural classification and clustering of web documents. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management* (pp. 359-368).

Joshi, S., Agrawal, N., Krishnapuram, R., & Negi, S. (2003, August). A bag of paths model for measuring structural similarity in web documents. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining* (pp. 577-582).

Levenshtein, V. I. (1966, February). Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady* (Vol. 10, No. 8, pp. 707-710).

Nebeling, M., Speicher, M., & Norrie, M. (2013, April). W3touch: metrics-based web page adaptation for touch. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 2311-2320).

Norrie, M. C., Nebeling, M., Di Geronimo, L., & Murolo, A. (2014, July). X-Themes: supporting design-by-example. In *International Conference on Web Engineering* (pp. 480-489). Springer, Cham.

Omer, B., Ruth, B., & Shahar, G. (2012). A New Frequent Similar Tree Algorithm Motivated by DOM Mining Using RTDM and its new variant–SiSTeR.

Reis, D. D. C., Golgher, P. B., Silva, A. S., & Laender, A. (2004, May). Automatic web news extraction using tree edit distance. In *Proceedings of the 13th international conference on World Wide Web* (pp. 502-511).

Tai, K. C. (1979). The tree-to-tree correction problem. *Journal of the ACM (JACM)*, *26*(3), 422-433.

Valiente, G. (2001, November). An Efficient Bottom-Up Distance between Trees. In *spire* (pp. 212-219).

W3C, "25 years ago the world changed forever," 2016. [Online]. Available: www.w3.org/blog/2016/08/25-years-ago-the-world-changed-forever/.

Xu, Z., & Miller, J. (2017). Estimating similarity of rich internet pages using visual information. *International Journal of Web Engineering and Technology*, *12*(2), 97-119.

Zanotti, M. (2016) Accessibility and Crowdsourcing: Use of semantic tags to improve web application accessibility (in Spanish), Univ. of La Plata, Argentina.

Zheng, S., Song, R., Wen, J. R., & Giles, C. L. (2009, November). Efficient record-level wrapper induction. In *Proceedings of the 18th ACM conference on Information and knowledge management* (pp. 47-56).

Zeng, J., Flanagan, B., & Hirokawa, S. (2013, June). Layout-tree-based approach for identifying visually similar blocks in a web page. In *2013 IEEE/ACIS 12th International Conference on Computer and Information Science (ICIS)* (pp. 65-70). IEEE.