Flexible Detection of Similar DOM elements

Julián Grigera[1,2,3], Juan Cruz Gardey[1,2], Gustavo Rossi[1,2], and Alejandra Garrido[1,2]

[1] LIFIA, Fac. de Informática, Univ. Nac. La Plata, La Plata, CP 1900, Argentina
[2] CONICET, Argentina
[3] CICPBA, Argentina
{juliang,jcgardey,gustavo,garrido}@lifia.info.unlp.edu.ar

Abstract. Different research fields related to the web require detecting similarity between DOM elements. In the field of information extraction, many approaches emerged to extract structured data from web documents, most of which require comparing sample documents to extract their underlying structure. Other fields of applicability like web augmentation or transcoding also require analyzing structural similarity, but on UI components with smaller structures than full documents, making them unsuitable for the algorithms generally used in information extraction. Instead, these approaches tend to rely on the DOM elements' location, but this does not resist structural changes in the document, and cannot locate similar elements placed in different positions. 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, together with a wrapper induction technique. We evaluated our algorithms with respect to other known approaches in the literature by comparing how they cluster a dataset of 1200+ DOM elements, using a manual clustering as ground truth. Results show that both proposed algorithms outperform all baseline ones. The proposed algorithms run in linear time, so they are faster than most approaches that analyze structural similarity.

Keywords: DOM · Information Extraction · Web Adaptation.

1 Introduction

Detecting similar elements in web interfaces is an important task in different fields of research. In Information Extraction's, the goal is to retrieve information from structured documents, which in turn requires analyzing similar documents for understanding their common underlying structure. In the Web Augmentation field, is usually necessary to detect similar elements in the web interfaces, so they can all be reached and modified in the same way. Both fields have naturally deviced different algorithms to determine whether 2 elements in the DOM (Document Object Model) are similar, which usually means that they share a common template.

Since information extraction generally focuses in retrieving data from full documents, these algorithms tend to analyze the structure of their DOM (Document Object Model) for comparison. Conversely, in web augmentation, there is more interest in detecting smaller DOM elements, so it is usual to use their location within the document to compare them. and rely less on their inner structure. For instance, a product page in an e-commerce application could typically be structured by a layout template (which defines a header, menu and footer) and a list of ratings, each defined by the same smaller-scaled template, as shown in Figure 1.

Using locators to establish similarity is quite effective for detecting elements that appear repeatedly in a same or different pages, but could present problems if these elements appear in different places, or when the outer structure suffer changes (hence, changing the elements' location).

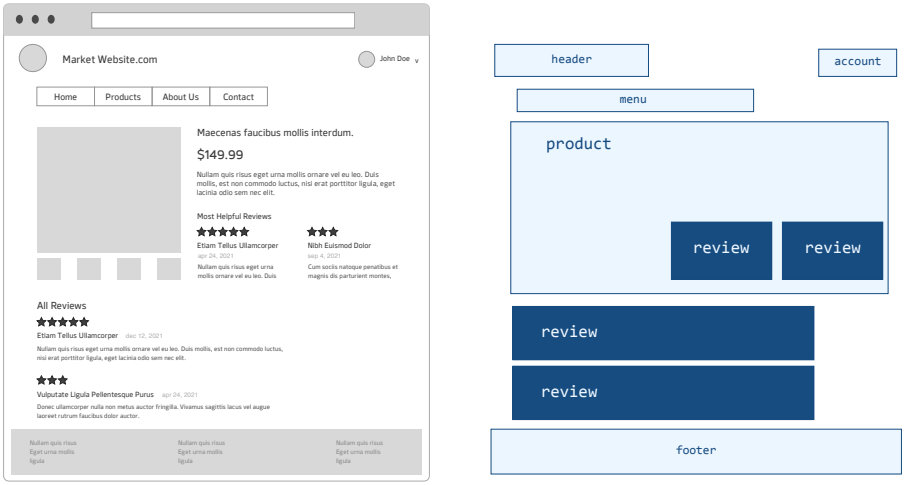

Fig. 1. A sample page with different templates. The review widget is highlighted.

1.1 DOM Structure Detection in Information Extraction

The field of Information Extraction aims at retrieving structured information from web applications, which requires to understand the semantics of the data. In the example shown in Figure 1, it is important to tell apart the rating text from the author and the date of submission. This is usually achieved by analyzing the underlying structure of web documents. Given that information extraction is generally performed on repeating structures (e.g. all product pages in an e-commerce website), this field usually resorts to find the underlying HTML

template, since most data intensive websites generate their content dynamically. In consequence, several approaches were devised to discover such templates by studying the structure of many similar pages [25].

A common way of doing this consists in clustering functionally equivalent components and then designing cluster-specific mining algorithms [24, 17]. These approaches typically produce a *wrapper*, which can be though of as a reverse-engineered template. Once the similar pages are grouped in clusters, the wrapper is generated by identifying the common structure within each cluster [17]. Wrappers are then used to analyze similar pages and extract their raw contents, in a similar way than regular expressions can be used to extract information from structured text.

Several algorithms used to cluster similar pages compute **tree edit distance** between the DOM structure of entire pages [24, 25]. 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 [4]. Other approaches in the area of Web Engineering even use variants of Web Scraping to make the application development process easier [23].

1.2 DOM Structure Detection in Web Augmentation

Web augmentation is another area that requires understanding DOM structure in order to externally modify web interfaces to pursue different goals like personalization or adaptation. However, in these cases, smaller elements are detected as opposed to full documents. Among the research works in this area we may find techniques like adaptation mechanisms for touch- operated mobile devices [22], transcoding to improve accessibility [2], augmentation to create personalized applications versions [5] and our own work on refactoring to improve usability [14, 15] and accessibility [9]. For the sake of conciseness, in this article we will call all these techniques *web adaptations*.

A widely used technique for detecting repeating HTML structures is the analysis of the DOM structure of the web interface. Viewing the page as a collection of HTML tags organized in a tree structure allows having a unique locator for each single element, known as XPath. Using an XPath, it is relatively easy to tell equivalent elements that belong in a repeating structure, like items on a list (`<li>` within a `<ul>`), or simple repetition of generic `<div>` tags. This technique has been widely used [1, 12] and it usually works well for detecting similar DOM elements, but since it depends exclusively on their locations within the HTML structure, it is not resilient to changes in the structure (which can happen over time) or the fact that equivalent elements could be located at different positions in the same page (or even different pages). In the example presented in Figure 1, the product page has a list of ratings at the bottom, but there is also a short list of the 2 most helpful ratings at the top. In this case, relying only on the XPath of the rating widgets would lead to incorrect results.

1.3 Contributions

The work presented in this paper intends to overcome the issues of both families of algorithms previously described, i.e. internal structure comparison (like tree edit distance) and path comparison (like XPath analysis). For this purpose we propose an algorithm that combine the strenghts of both approaches, with enough flexibility to prefer one over the other depending on the situation. This means that, when needing to compare large DOM elements, it should rely on inner structure comparison, but when these elements have few inner nodes, and hence the risk of grouping dissimilar elements is higher, it should switch to comparing their location within the document.

In this paper we describe a web widget comparision algorithm that's designed to adapt to differently sized DOM elements. 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 shown, which considers also on-screen dimensions and position of the elements. 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. These algorithms, namely **Scoring Map** and **Scoring Map Dimensional Variant** were first presented on a previous article [13], and in this work we extend it in different directions.

More specifically, the contributions of this paper are the following:

- We extend the related work section with other works that are relevant to our proposal.
- We describe the Scoring Map and its dimensional variant algorithms with greater detail, including improvements on their past implementation, and the rationale behind the algorithm's design.
- We show how we optimized the algorithm's parameters in order to get better results.
- We extend the previous experiment with new samples and analysis, and evaluate the performance of the algorithms with an additional measure.

2 Related Work

Detecting DOM element similarity has been covered in the literature in the context of different research areas, like web augmentation or web scraping. Because of the broad applicability of this simple task, many different techniques have emerged. 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 [4].

In this section we describe many of these works, and organize them in three main groups: tree edit distance algorithms, bag of paths methods, and other approaches. We also briefly describe some other algorithms that serve the same purpose but take radically different approaches.

2.1 Tree Edit Distance Algorithms

Since DOM elements can be represented as tree structures, they can be compared with similarity measures between trees, such as edit distance. This technique is a generalization of string edit distance algorithms like Levenshtein's [18], 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 of such algorithms was proposed by Tai [26], and it 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) [25], which uses mappings such as Tai's but with restrictions on the mappings that make it faster. Building on RTDM, Omer et al. [24] 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 number 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 [27], but RTDM and its variants are best 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 [11]. 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. Fard and Mesbah also developed a variant of the TED algorithm to calculate the diversity of two DOM trees [7], in which the amount of edit operations are normalized by the number of nodes of the biggest tree.

TED algorithms have also been proposed to compare entire web pages [28]. This work defines the similarity of two web pages as the edit distance between their block trees, which are 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 [17], with a variant for the specific case of XPaths in HTML documents. The latter was used in different approaches for documents clustering [12].

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(n2N), 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 [3]. This work takes structurally rich subtrees and represents them as pq-grams, which 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 one 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 Locator-based comparison

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 [12] use inbound links to determine similarity). The reason is 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 key role. Amagasa, Wen and Kitagawa [1] 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 [31]. 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 that contain them.

2.4 Other Approaches

Different approaches have been developed that have little or no relation with tree edition distance or paths comparison. Many approaches like the one proposed by Zeng et. al [30], 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 [16], enabling a fast way of comparing documents without the need of traversing 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.

String-based comparison have also been proposed to find differences and similarities between DOM trees. The work of Mesbah and Prasad uses a XML-differencing tool to detect DOM-level mismatches that have a visual impact on web pages rendered on different browsers [21]. Moreover Mesbah et al., developed a DOM tree comparison algorithm to derive the different states of a target UI to support automatic testing of AJAX applications [20]. This comparison is based on a pipeline of "comparators", in which each comparator eliminates specific parts of the DOM tree (such as irrelevant attributes). At the end, after all the desired differences are removed, a simple string comparison determines the equality of the two DOM strings.

3 Proposed Algorithm

We devised an algorithm that has the capability of comparing DOM elements of different sizes. This flexibility is achieved considering both the element's internal structure and location within the DOM tree. As the inner structure grows larger, the more relevant it becomes in the final score. Conversely, when the structure is small, the location path gains more relevance. This design makes the algorithm actually prefer the inner structure when possible, since it is a better strategy to resist DOM changes or compare differently located but equivalent elements. It will rely on the element's location when the inner structure size is insufficient to tell DOM elements apart. We also introduce a variant that considers also the elements' dimensions and position as rendered on the screen.

The algorithm generates a dictionary-like structure (i.e. a map) for each DOM element, which contains structural information through its HTML tags and attributes (keys) and relevance scores (values). These maps can be compared to obtain a similarity score between any two DOM elements. This map, and the comparison algorithm are both used for a wrapper induction technique, which is important for different applicabilities of the algorithm.

3.1 Rationale

The main idea behind the algorithm is to compare elements' structure and location, in terms of inner HTML tags and path from the document's root, respectively. In order to do this, it first captures both structure and location of each element to compare, as shown in Figure 2

Regarding inner structure comparison, the algorithm looks for matches in the tags that are at the same level. For instance, if the root tag of both elements being compared is the same, that counts as a coincidence. If a direct child of the root matches a direct child of the other element, it also counts as a coincidence, but to a lesser degree than the root element. This degree is defined by a score that decreases as the algorithm moves farther away from the root towards the leaves. The criterion behind this decision is that equivalend DOM elements usually present some kind of variability, e.g. two product panels may come from the same template, but only one of them shows a discount price. This variability

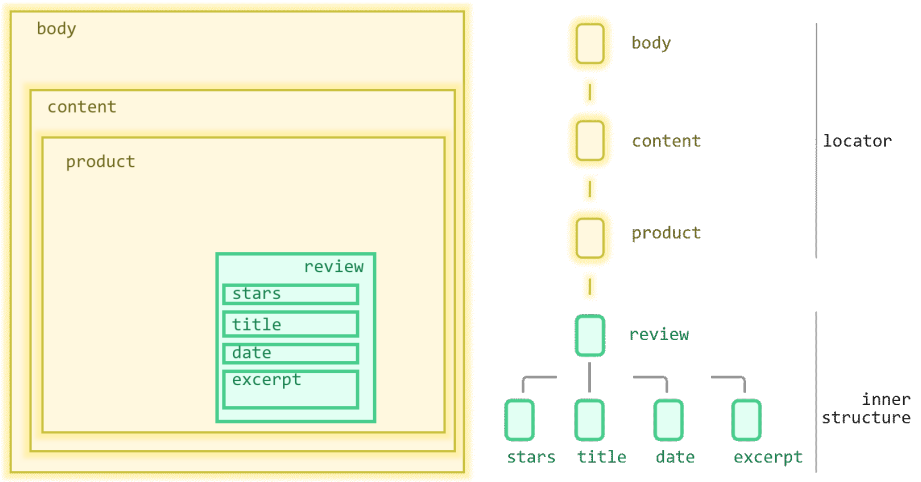

Fig. 2. A schematic of a DOM element as seen in its containing document (left), and the catured data for comparison in terms of location and structure (right).

tends to happen farther from the root of the element, so the algorithm was designed for this to have a lesser impact in the comparision, making it resistent to small variations but sensitive to more prominent ones.

Inner structure comparison doesn not only compare tag names, but it also considers tag attributes. Attributes are only captured by their name, and not their values, since their presence and number is usually enough to determine equivalent tags. There is only one exception, the `class` attribute, since it's abundantly present in many tags, and its values are usually shared between equivalent DOM elements that, in this case, makes them more useful than the simple tag presence. It is worth mentioning that the `id` attribute's value would not be helpful in this case, since this value is meant to be unique in the document, hence hindering the algorithm's ability to match equivalent elements that will naturally have different ids.

The outer comparison, which is the comparison of the elements' paths, is similar to the inner part. The main difference is that the way to the root of the document is a list and not a tree, so there is exactly one tag at each level.

3.2 Scoring Map Generation

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 [24].

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 Figure 3.

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.

In the original algorithm [13] the initial score for the parent node was generated inversely proportional to the height of the tree. The rationale bethind this is to give the algorithm the flexibility to detect similar elements relying less on their location when the trees are large enough, so the inner structure scores get more weight on the overall comparison. In the new algorithm, we simply allow

Fig. 3. A DOM element along with its generated scoring map. Yellow indicates outer path and green indicates inner structure.

any initial relevance, both outer and inner. This still keeps the same properties, since the number of levels determine the amount of key/value tuples that will populate the map. The taller the inner tree is, the more values in the map to represent it. Conversely, when the tree is shorter, the external path entries gain more relevance. This design made the algorithm simpler, while keeping its flexibility.

Another distinctive aspect of the map structure is the elements' limited knowledge of their tree structure. 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 results obtained in 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.

3.3 Maps Comparison

Once there is a map for the 2 elements to be compared, they are used to get a score that's based on the values at each level. 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.

Intuitively, this function compares similitude between maps by their common keys with respect to the total number of combined keys. This is similar to the Jaccard index for sample sets, which calculates the ratio between intersection and union. This makes sense for comparing maps, since for comparison purposes they can be seen as sets of keys - that cannot be repeated. The way the Bag of Paths algorithm calculates similarity in the same way [17].

3.4 Scoring Map with Dimensional Awareness

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 relies on geometric properties 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.

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 Figure 4.

Fig. 4. Dimensional alignment example with 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.

3.5 Time Complexity

The proposed 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). Since stages of the comparision, i.e. the generation of the scoring map and the maps comparison, happen in linear time, we can conclude that the algorithm runs in linear time too.

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.

3.6 Wrapper Induction

Being information extraction one of the potential areas of applicability of our algorithm, it is important to be able to generate a matching template for the elements being "scraped", i.e. a *wrapper*.

We developed a way of generating wrappers with the same maps that we use for comparing elements. A first element is taken as reference to generate a base map the same way we explain in section 3.2. 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 3.2. 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 exceeds 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 value that 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. 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 Validation

In order to test the efficacy of our algorithm, we carried out an experiment using DOM elements from several real websites. The main objective was to compare the ability of the algorithm to group together DOM elements that are equivalent, that is, show the same kind of structured data, or serve the same purpose (in the case of actionable elements such as menu items or buttons). We compare the algorithm's performance with 3 others used as baselines, using a manual clustering as reference.

A similar experiment is described in our previous work [13], but we have extended the number of samples and cases. We also show different post-hoc analyses that offer new insights of the results. All the resources for replicating the validation are available online[4].

4.1 Dataset Generation

We created a dataset with small and medium DOM elements from real websites. To do this, we first had to create a tool for capturing the elements, and then generate the reference groups. The tool allows for selecting DOM elements from any website with the help of a highlighter to accurately display the limits of each element, and grouping them together for generating a reference cluster. We implemented our tool as a GreaseMonkey[5] script, with a Pharo Smalltalk backend[6], storing the elements in a MongoDB database.

Using our tool, we captured **1204** DOM elements grouped in **197** reference clusters from a total **54** websites. The elements ranged from medium sized widgets like forum comments or item presentations (such as products) to smaller components like menu items, buttons or even atomic components like dates

[4] https://github.com/juliangrigera/scoring-map
[5] https://addons.mozilla.org/en-GB/firefox/addon/greasemonkey/
[6] https://pharo.org

within larger elements. We also aimed at capturing equivalent elements placed in different locations of the page to represent the situations where a template is reused, but it was also a good approximation for the scenario where the document DOM structure changes over time. These, however, were a small proportion with respect to the total dataset.

Clustering comparison is not a trivial task, and different metrics can favor (or fail at) particular scenarios. For this reason, we selected 2 different metrics to evaluate all the algorithms. The first one, also used in the original experiment, is f-score, which is based on a precision/recall analysis of counting pairs. The second analysis is an asymmetric index based on cluster overlapping.

4.2 Preparation

We implemented 3 algorithms from the literature to use as baseline, each one representing a family of algorithms covered in Section 2. For the tree edit distance algorithms, we chose RTDM [25], for being more performant than starndard tree edit distance, and having available pseudocode. In the bag of paths area, we chose the Bag of XPath algorithm [17]. Finally, for the locator-based algorithms we implemented a straightforward XPath comparison, which is the root of many of such algorithms, which usually add variations. 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:

```
/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.

After implementing the algorithms, we performed parameter optimization: almost all the algorithms required a similarity threshold to be set. XPath comparision was the only exception, since it does not produce a similarity score but a boolean result, the clustering procedure does not depend on the order in which the elements are fed. Additionally, the Scoring Map algorithm requires two initial scores for creating the maps of the elements to be compared: one for the root of the target element (inner relevance), and the other one for the parent node (outer relevance).

With the aim of finding the parameters values that give the highest performance (in terms of f-score described in next section), we performed a bayessian hyperparameter optimization. Bayessian methods can find better settings than random search in fewer iterations, by evaluating parameters that appear more promising from past iterations.

We used HyperOpt Python library[7] which requires four arguments to run the optimization: a search space to look for the scores, an objetive function that takes in the scores and outputs the value to be maximized, and a criteria to select the next scores in each iteration. The search space for both the inner and outer relevance was an interval between 0 and 100 uniformly distributed, while the space for the threshold was an interval between 0 and 1. The objetive

[7] https://github.com/hyperopt/hyperopt/

function runs the maps comparison with the selected threshold and relevance scores in the target dataset and returns the resulting f-score negated, because the optimization algorithm expects from the objetive function a value to minimize. Lastly, the criteria to select the scores is Tree of Parzen Estimators (TPE).

We ran the optimizer perfoming 100 iterations over the search space for each of the implemented algorithms. The highest reached f-score for the Scoring Map was 0.9916, with *outer relevance*=39.65, *inner relevance*=57 and *threshold*=0.7. Regarding RTDM and Bag of XPath, the highest f-score was 0.338 and 0.339 with the thresholds 0.8 and 0.54 respectively.

4.3 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 20 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.

We compared all the clusterings with the manual one. For the f-score analysis we used pair comparison: since clusterings are not simple classification problems with predefined classes, there is a special interpretation for calculating precision and recall. Given two elements:

- if both the automated and reference clustering place them in the same group, then we consider this case a **true positive**.
- if both the automated and reference clustering place them in different groups, then we consider this a **true negative**
- if both elements are in the same group in the reference clustering but not in the automated clustering, we consider it a **false negative**
- if 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:

$$F1 = 2 * \frac{precision * recall}{precision + recall}$$

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

In addition to the f-score measure, we calculated an overlapping baser measure proposed by Meilă and Heckerman [19], which is relevant to our work since all baseline clusterings are compared to a reference clustering taken as ground truth. The formula used is the following:

$$MH(C, C') = 1/n \sum_{i=1}^{k} max_{C'_j \in C}(|C_i \cap C'_j|)$$

where C' is the manual reference clustering, and C is the clustering automatically generated by the algorithm. In the rest of the paper we will refer to this measure as *MH*.

4.4 Results

Results are shown in Table 1 and Figure 5. Regarding the f-score analysis, it is important to remark that, the precision/recall and f-score values are averaged from a set of 20 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 necessarily match the values on the f-score column.

Table 1. Evaluation results, including number of false positives and negatives for f-score analysis.

Algorithm	F. Positives	F. Negatives	Precision	Recall	F-Score	MH
Bag of XPaths	17503	380	0.2070	0.9231	0.3863	0.3382
RTDM	17060	467	0.2080	0.9056	0.3384	0.4088
XPaths	1252	420	0.7834	0.9151	0.8442	0.5780
Scoring Map	0	358	1.0	0.9276	0.9625	0.8219
Scoring Map (D)	0	436	1.0	0.9117	0.9538	0.7847

There is a pronounced difference both in f-score and MH measure, between the algorithms that consider the elements' inner structure, more prepared for comparing full documents, and the algorithms that consider the location of the elements, namely XPath, Scoring Map and Scoring Map dimensional variant.

Both Scoring Map and Scoring Map dimensional variant outperformed the baseline ones in f-score and MH measure. Regarding precision and recall, our algorithms got perfect precision, while shared a similar recall value with all baseline algorithms (although Scoring Map was still the highest). In this context, high precision means lower false positives ratio, which 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.

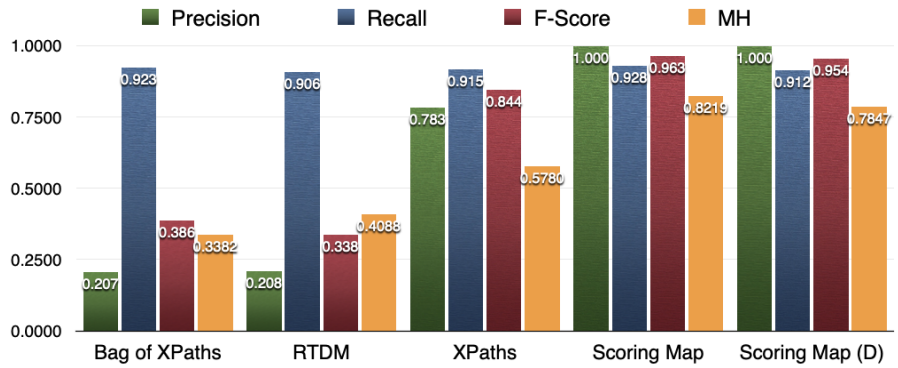

Fig. 5. Evaluation results. Precision, Recall and F1 Score, and Meilã-Heckerman measure.

The worst performing algorithms in this analysis showed high recall values, but when precision drops to low levels (below 0.3) as shown in Table 1.

While still outperforming all baseline algorithms, the dimensional variant of the Scoring Map algorithm got slightly lower results than the original algorithm. By examining the samples, we could observe that the dimensional variant worked well in some particular cases where the structure was different but the spatial properties similar (e.g. a menu item that has a submenu in a list where its siblings don't have any).

Since the results were averaged, we analyzed the Standard Deviation to assess the variations in the different orders of elements, but we found it to be very low in all cases. The standard deviation in the f-score for the resulting clusters depending on the order was 0.0006 for RTDM, 0.0003 for bag of XPaths, 0.0002 for our scoring map algorithm and 0.0004 for the dimensional variant. With MH measures, it was also very low: 0.0071 for RTDM, 0.0021 for Bag of XPaths, 0.0022 for Scoring Map and 0.0045 for the dimensional variant.

4.5 Post-Hoc Analysis

The dataset allowed us to make different analyses, since there is more data in the captured DOM elements than their structure.

The first analysis, also present in our previous work, is the analysis per element size, defined by the height of the DOM tree. This is especially relevant to our proposal because it allows to validate the alleged flexibility of the Scoring Map algorithm. In line with the previous experiment, we observed that the baseline algorithms focused on inner structure 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 Figure 6.

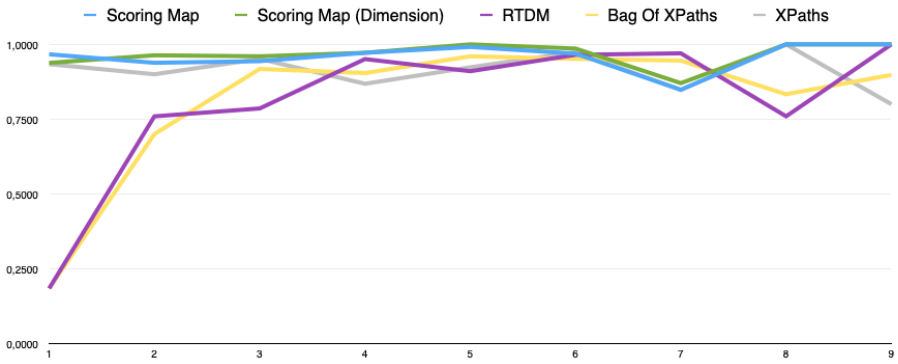

Fig. 6. F-score results by DOM element height for each algorithm.

The second analysis consisted in restricting the groups of DOM elements by their domain. We were interested in this analysis since working on a single application is a very common use case for all the applicabilities previously listed. Results show a major improvement across all algorithms, especially those performing poorly in the general analysis. This is most likely due to the reduced search space. Both our algorithms still outperform all baseline ones, although in this particular study the dimensional variant is the best one for a small margin. These results can be seen in Figure 7.

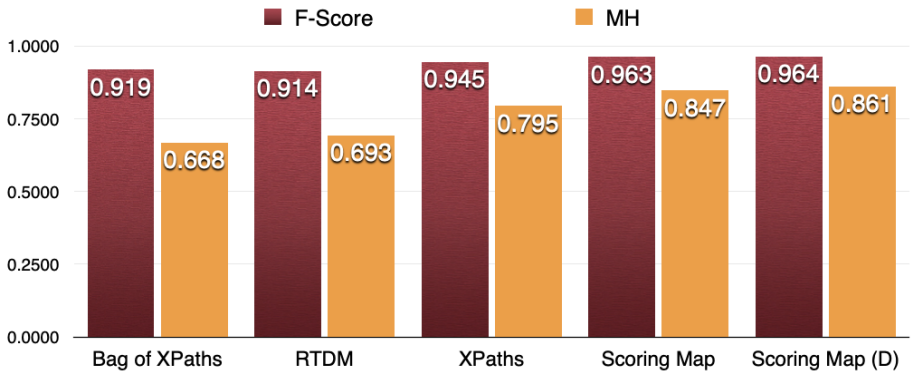

Fig. 7. Evaluation results of restricted clustering by domain. F1 Score and Meilã-Heckerman measure.

4.6 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 4.3.

5 Use Cases

We used our algorithms to measure structural similarity in the context of three web adaptation approaches in the specific areas of UX and accesibility. These approaches are described in the next subsections.

5.1 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 [8]. 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 the refactoring notion applied to external quality factors, like usability [6] or accessibility [9]. 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 [10].

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 [15]. 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) [9], 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).

5.2 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 [29] 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. 8, where the threshold is adapted to include or exclude video lists in the clustering. 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.

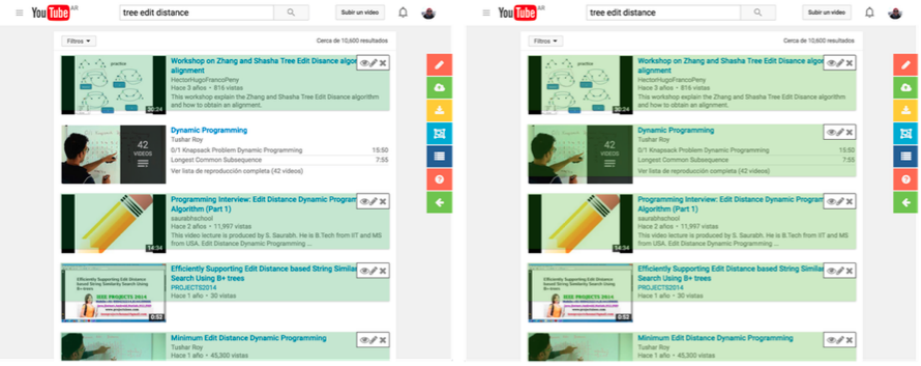

Fig. 8. Web Accessibility Transcoder capturing similar DOM elements, using two different threshold values.

5.3 UX-Painter

UX-Painter is a web-extension to apply CSWRs on a web page with the aim of improving the user experince. These transformations are meant to perform small changes on the user interface (UI) without altering the application functionality. The tool is intended for UX-designers, who may not have programming skills, to be able to freely excercise design alternatives directly on the target application. CSWRs are applied without the need of coding the changes, by selecting the target elements on the UI and configuring specific parameters, and can be saved in different application versions which then can be re-created to perform UX evaluations such as user tests or inspection reviews.

On a first version of the tool, each refactoring was applied on a single UI element on a specific page. Later, by conducting interviews with professional UX designers, we discovered that it was important to allow applying a refactoring to multiple instances of a UI element that can be spreaded across different pages, since it is common to re-use certain UI elements in different parts of the target application. In this way, we improved UX-Painter with Scoring Map to generate a wrapper for each element refactored by the user in order to identify similar elements to apply the same transformation on. Finding similar elements with

the wrapper induction technique is more flexible and robust than using XPath expressions because the elements location can differ in each case, and because XPath expressions tend to break as web pages evolve.

6 Conclusions and Future Work

In this paper we have presented an algorithm to compare individual DOM elements, along with a variant that uses the elements' dimensions and positioning. Both algorithms are flexible enough to successfully compare DOM elements of different sizes, overcoming limitations of previous approaches. We also presented a wrapper induction technique that's based on the comparison algorithm's implementation.

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. Both algorithms run in order O(n), being n the total number of nodes of both trees added together.

Through a validation with 1200+ DOM element, we compared our proposal with others from the literature, spanning different strategies. We compared the clusterings generated by each algorithm against a ground truth, and used two different metrics to compare them. The results showed that our algorithms outperform all the baseline algorithms according both metrics.

We performed other analyses derived from the evaluation: one showed the flexibility of our algorithm, which was able to keep a stable performance across different levels (heights) of DOM elements, and the other demonstrated the adjusted performance of all algorithms when the clustering was restricted by application domain.

Another relevant extension to the validation could be testing the resistance of the algorithm to DOM structure changes. This is, however very complicated to set up because of the difficulty of gathering realistic samples of elements that change places over time. A controlled experiment with generated test data could be carried out, but it would have low external validity.

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

References

1. Amagasa, T., Wen, L., Kitagawa, H.: Proximity search of xml data using ontology and xpath edit similarity. In: International Conference on Database and Expert Systems Applications. pp. 298–307. Springer (2007)
2. Asakawa, C., Takagi, H.: Web accessibility: A foundation for research, chapter transcoding (2008)

3. Augsten, N., Böhlen, M., Gamper, J.: Approximate matching of hierarchical data using pq-grams. In: Proceedings of the 31st international conference on Very large data bases. pp. 301–312 (2005)
4. Buttler, D.: A short survey of document structure similarity algorithms. Tech. rep., Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States) (2004)
5. Díaz, O.: Understanding web augmentation. In: International conference on web engineering. pp. 79–80. Springer (2012)
6. Distante, D., Garrido, A., Camelier-Carvajal, J., Giandini, R., Rossi, G.: Business processes refactoring to improve usability in e-commerce applications. Electronic Commerce Research **14**(4), 497–529 (2014)
7. Fard, A.M., Mesbah, A.: Feedback-directed exploration of web applications to derive test models. In: 2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE). pp. 278–287 (2013). https://doi.org/10.1109/ISSRE.2013.6698880
8. Fernandez, A., Insfran, E., Abrahão, S.: Usability evaluation methods for the web: A systematic mapping study. Information and software Technology **53**(8), 789–817 (2011)
9. Garrido, A., Firmenich, S., Rossi, G., Grigera, J., Medina-Medina, N., Harari, I.: Personalized web accessibility using client-side refactoring. IEEE Internet Computing **17**(4), 58–66 (2012)
10. Garrido, A., Rossi, G., Distante, D.: Refactoring for usability in web applications. IEEE software **28**(3), 60–67 (2010)
11. Griazev, K., Ramanauskaitė, S.: Html block similarity estimation. In: 2018 IEEE 6th Workshop on Advances in Information, Electronic and Electrical Engineering (AIEEE). pp. 1–4. IEEE (2018)
12. Grigalis, T., Čenys, A.: Using xpaths of inbound links to cluster template-generated web pages. Computer Science and Information Systems **11**(1), 111–131 (2014)
13. Grigera, J., Gardey, J.C., Garrido, A., Rossi, G.: A scoring map algorithm for automatically detecting structural similarity of dom elements (2021)
14. Grigera, J., Garrido, A., Panach, J.I., Distante, D., Rossi, G.: Assessing refactorings for usability in e-commerce applications. Empirical Software Engineering **21**(3), 1224–1271 (2016)
15. Grigera, J., Garrido, A., Rivero, J.M.: A tool for detecting bad usability smells in an automatic way. In: International Conference on Web Engineering. pp. 490–493. Springer (2014)
16. Hachenberg, C., Gottron, T.: 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 (2013)
17. Joshi, S., Agrawal, N., Krishnapuram, R., Negi, S.: 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 (2003)
18. Levenshtein, V.I., et al.: Binary codes capable of correcting deletions, insertions, and reversals. In: Soviet physics doklady. vol. 10, pp. 707–710. Soviet Union (1966)
19. Meilă, M., Heckerman, D.: An experimental comparison of model-based clustering methods. Machine learning **42**(1), 9–29 (2001)
20. Mesbah, A., van Deursen, A., Roest, D.: Invariant-based automatic testing of modern web applications. IEEE Transactions on Software Engineering **38**(1), 35–53 (2012)

21. Mesbah, A., Prasad, M.R.: Automated cross-browser compatibility testing. In: Proceedings of the 33rd International Conference on Software Engineering. p. 561–570. ICSE '11, Association for Computing Machinery, New York, NY, USA (2011)
22. Nebeling, M., Speicher, M., Norrie, M.: W3touch: metrics-based web page adaptation for touch. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. pp. 2311–2320 (2013)
23. Norrie, M.C., Nebeling, M., Geronimo, L.D., Murolo, A.: X-themes: supporting design-by-example. In: International Conference on Web Engineering. pp. 480–489. Springer (2014)
24. Omer, B., Ruth, B., Shahar, G.: A new frequent similar tree algorithm motivated by dom mining using rtdm and its new variant–sister (2012)
25. Reis, D.D.C., Golgher, P.B., Silva, A.S., Laender, A.: Automatic web news extraction using tree edit distance. In: Proceedings of the 13th international conference on World Wide Web. pp. 502–511 (2004)
26. Tai, K.C.: The tree-to-tree correction problem. Journal of the ACM (JACM) **26**(3), 422–433 (1979)
27. Valiente, G.: An efficient bottom-up distance between trees. In: spire. pp. 212–219. Citeseer (2001)
28. Xu, Z., Miller, J.: Estimating similarity of rich internet pages using visual information. International Journal of Web Engineering and Technology **12**(2), 97–119 (2017)
29. Zanotti, M.: Accessibility and crowdsourcing: Use of semantic tags to improve web application accessibility. Univ. of La Plata, Argentina (2016)
30. Zeng, J., Flanagan, B., Hirokawa, S.: 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 (2013)
31. Zheng, S., Song, R., Wen, J.R., Giles, C.L.: Efficient record-level wrapper induction. In: Proceedings of the 18th ACM conference on Information and knowledge management. pp. 47–56 (2009)