This is the Title of Your Thesis

Sacha Brisset, *Romain* Rouvoy, *Lionel* Seinturier, *Renaud* Pawlak

Submitted in partial fulfilment
of the requirements for the degree of

Master of Science

Department of Computer Science
Brock University
St. Catharines, Ontario

Abstract

This is the thesis abstract. It should be a maximum of 350 words. It should summarize the content of your thesis. This includes the main results that you obtained.

Acknowledgements

This is where you acknowledge all those who have helped you. This usually includes your supervisor, thesis committee members, staff that helped with hardware and software support, fellow students, parents and other loved ones, and Kevin Bacon.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This is the introduction chapter. This is fill text to fill up a page. This is fill text to fill

up a page. This is fill text to fill up a page. This is fill text to fill up a page. This is fill text to fill up a page. This is fill text to fill up a page. This is fill text to fill up a page. This is fill text to fill up a page. This is fill text to fill up a page. This is fill text to fill up a page. This is fill text to fill up a page.

Chapter 2

Tree Matching

2.1 abstract

Tree matching techniques have been investigated in many fields, including web data mining and extraction, as a key component to analyze the content of web pages. However, when applied to existing web pages, traditional tree matching approaches, covered by algorithms like *Tree-Edit Distance* (TED) or *XyDiff*, either fail to scale beyond a few hundred nodes or exhibit a relatively low accuracy.

In this article, we therefore propose a novel algorithm, named *Similarity-based Flexible Tree Matching* (SFTM), which enables high accuracy tree matching on real-life web pages, with practical computation times. We approach tree matching as an optimisation problem and leverage node labels and local topology similarity in order to avoid any combinatorial explosion. Our practical evaluation demonstrates that SFTM significantly improves the state of the art in terms of accuracy, while allowing computation times significantly lower than the most accurate solutions. By gaining on these two dimensions, SFTM therefore offers an affordable solution to match complex trees in practice.

2.2 Introduction

The success of Internet has led to the publication and the delivery of a deluge of structured content. Nowadays, web services and applications are heavily adopting tree-based documents to structure and transfer online content. However, these web pages keep evolving over time and keeping track of such changes remains a critical issue for the ecosystem and the research community. Examples of usages that require

to detect or track changes in web pages include web extraction [?, ?, ?], web testing [?, ?], comparison of web service versions [?], web schema matching [?], and automatic re-organization of websites [?].

To date, few solutions are specifically designed or tested to match and compare two web pages. However, the more general question of tree matching has been extensively studied by two families of solutions applicable to the problem of web page matching: 1. *Tree Edit Distance* (TED) [?] and TED-related solutions, and 2. XML differentiation (diff) solutions.

TED is the first and most widely known approach to match trees. The matchings computed by TED solutions are optimal and there have been much effort into developing openly available efficient implementations of the algorithm [?, ?, ?]. Despite these efforts, TED remains costly to compute. A recent study [?] theoretically showed that no algorithm could compute the optimal TED in less than $O(N^3)$ worst time complexity. To address TED's limitations, several restrictions to TED have been developed. These TED-related algorithms add constraints to the produced matching allowing to trade accuracy for speed.

XML diff solutions aim to find the sequence of editions between two XML trees. The approach is similar to TED, but solutions sometimes make use of XML specificities. For example, the most widely-known XML diff solution—XyDiff [?]—is extremely fast, but makes heavy use of XSD schemas and XML primary keys, which cannot be assumed on for any web page. Without such an additional information, the algorithm unfortunately yields low-accuracy results.

Overall, when matching two web pages, even the most efficient TED implementation [?] offers far from optimal accuracy (69% of precision in average in our empirical evaluation) for computation times often reaching several seconds. The lack of accuracy may be due to the restrictions TED solutions impose on the produced matching: ancestors and siblings order must be preserved. However, such restrictions do not hold for web pages and Figure ?? illustrates how TED can report biased matchings, even on simple trees.

To address these restrictions when attempting to match two web documents, [?] extended TED with some additional move operations executed *a posteriori* to address the ancestry restriction and [?, ?] developed her own *Flexible Tree Matching* (FTM) algorithm to address the ancestry restriction problem. Unfortunately, while FTM provides a truly restriction-free matching, its high complexity does not allow FTM to scale beyond more than a few dozens of nodes, which is far below the average size of real-life web pages.

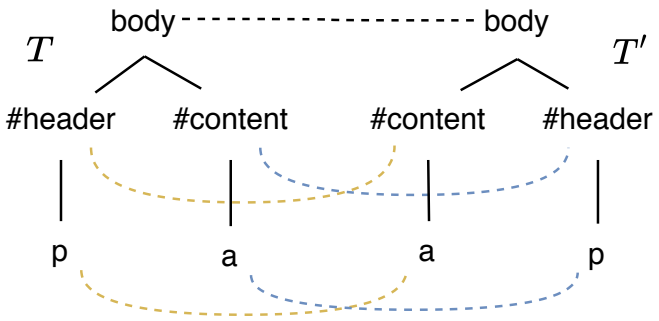


Figure 2.1: Example of matching biased by the TED (as computed by APTED).

In the line of the aforementioned work, this article therefore aims at enabling the fast and non-restricted comparison of complex web pages. In particular, we propose an alternative to the state-of-the-art FTM algorithm, named *Similarity-based Flexible Tree Matching* (SFTM), that leverages similarity metrics to speed up the comparison of complex trees. SFTM shares the properties of FTM to offer a non-restricted tree matching, while offering computation times much lower than FTM, even on restricted versions of the problem. To match two web page trees, the approach taken by SFTM strongly differs from traditional techniques. In particular, existing matching algorithms are *structure-centric*: they leverage the structure of both trees to select the nodes to visit and compare. SFTM instead relies on a *label-centric* approach: it prunes the space of possible matchings using nodes' *label* and considers the tree topology *a posteriori* to propagate information contained in the nodes.

We compared SFTM to other state-of-the-art solutions on a large dataset of popular web pages. SFTM showed almost twice more efficiency than the best existing solution. Overall, our algorithm SFTM allows to consistently match real-life web pages with high precision (89% precision on average) in reasonable time (182 ms on average).

The code for both SFTM and its benchmark is available openly.[1]

The remainder of this article is organized as follows. Section **??** and **??** cover related work, with section **??** focusing in details on the *Flexible Tree Matching* (FTM) original algorithm. Section **??** presents *Similarity-based Flexible Tree Matching* (SFTM), our extension of FTM that leverages the node labels and local topology similarity to guide the comparison. Section **??** thoroughly evaluates our solution against the state of the art on a realistic dataset of web documents. Section **??** discusses the threats to validity of our contribution. Section **??** concludes and overviews

[1] https://anonymous.4open.science/r/7ae57bd7-3b29-463a-88a4-d31c04ecfcd2/

some perspectives for this work.

2.3 Related Work

***Tree Edit Distance* (TED)** Comparing two trees is a problem that has been at the center of a significant amount of research. In 1979, Tai [?] introduced the *Tree Edit Distance* (TED) as a generalization of the standard *edit distance* problem applied to strings. Given two ordered labeled trees T and T', the TED is defined as the minimal amount of node insertion, removal or relabel to transform T into T', while different cost coefficients can be assigned to each type of operation. By following an optimal sequence of operations applied to T, it is possible to match the nodes between T and T'. This problem has been extensively studied since then to reduce the space and time complexity of the algorithm that computes the TED. To the best of our knowledge, the reference implementation available today is the *All-Path Tree Edit Distance* (APTED) [?, ?, ?] with a complexity of $O(n^2)$ in space and $O(n^3)$ in time in the worst case, where n is the total number of nodes ($n = |T_1| + |T_2|$). In our work, we consider APTED as one of the baselines to evaluate our contribution.

[?] showed that TED cannot be computed in worst-case complexity lower than $O(n^3)$. In order to circumvent this limitation, several restricted versions of the TED problem have been formulated. The *Constrained Edit Distance* [?, ?] is an edit distance where disjoint subtrees can only be mapped to disjoint subtrees. The *Tree Alignment Distance* [?] is a TED where all insertions must be performed before any deletion. The *Top-Down* distance [?] is computable in $O(|T| \times |T'|)$, but imposes as a restriction that the parents of nodes in a mapping must be in the mapping. The *Bottom-Up* distance [?] between trees builds a mapping in linear time, but such mapping must respect the following constraint: if two nodes have been mapped, their respective children must also be part of the mapping. [?] proposes a variation of the *Top-Down* mapping, called *Restricted Top-Down Mapping* (RTDM), where replacement operations are restricted to the leaves of the trees, which delivers considerable speed gains, despite a theoretical worst case time complexity still in $O(N^2)$. By definition TED already sets strong restrictions on produced matchings: sibling order and ancestry relationships must be preserved [?]. These restrictions are particularly problematic when matching two full web pages together [?]. While above solutions improve computation times, they answer a restricted version of the TED problem leading to an even more restricted set of possible matchings.

XML diff While TED-related approaches focus on computing a distance between trees, another part of the scientific literature focuses on inferring the set of edit operations between two XML documents. Most XML diff solutions use an intermediary matching step in order to compute the diff. Computing the set of diff from a given matching is quite straightforward, which means that most works on the subject actually focus on the matching part. XyDiff [?] matches and computes the diff of two XML documents very quickly. To do so, XyDiff hashes subtrees from both documents and prunes the space of matching possibilities by matching subtrees with identical hashes. The algorithm can also make use of id attributes and XSD schemas if they exist. On the other end of the spectrum, X-Diff [?] favors accuracy over speed and computes an optimal matching using hashings of path signatures. XKeyDiff [?] builds on XyDiff and adds matching logic based on XML primary keys, `XML_SIM_CHANGE` [?] and `XREL_CHANGE_SQL` [?] match XMLs stored in relational databases using SQL. Phoenix [?] interestingly uses a more flexible similarity metric between nodes (*e.g.*, to compare the content of two nodes, they use the *Longest Common Sequence*) and choose how to match each subtree by recursively applying the Hungarian algorithm [?]. Unfortunately, Phoenix runs in $O(n^4)$ and yields less accurate results than X-Diff. In our empirical evaluation, we evaluated our solution along with XyDiff, which is widely known and used for XML diff, has an efficient implementation openly available and runs in scalable computation times.

***Flexible Tree Matching* (FTM)** In [?], TED is found to be unpractical when applied on DOM, as the resulting matching enforces ancestry relationship—*i.e.*, once two nodes n, n' have been matched, the descendants of n can only be matched with the descendants of n', and *vice versa*. Consequently, Kumar *et al.* [?, ?] introduced the notion of *Flexible Tree Matching* (FTM), which relaxes the ancestry relationship constraint at the price of a stronger complexity. It restricts its use to small HTML trees composed of less than a hundred nodes, thus making it unpractical for modern web documents, often including more than a thousand nodes. Furthermore, to the best of our knowledge, there is no public implementation of FTM that can be considered for comparison.

We therefore aim at reducing the complexity of the FTM algorithm in order to scale on complex web pages without enforcing restrictions on produced tree-matching solutions. While all above contributions are structure-based, we build on FTM's approach and rather offer a flexible, label-based matching where labels are used to match nodes and structure is only used *a posteriori* to improve the matching.

Our contributions therefore read as follows:

1. we develop an algorithm inspired by FTM, coined as *Similarity-based Flexible Tree Matching* (SFTM), by leveraging the notion of label similarity, and similarity propagation to reduce the computation time, and
2. we apply mutations on real-life web documents to provide a thorough evaluation of our implementation of SFTM, showing that it outperforms state-of-the-art approaches in terms of efficiency.

2.4 Flexible Tree Matching (FTM)

The *Similarity-based Flexible Tree Matching* (SFTM) we introduce in this article can be considered as an extension of the *Flexible Tree Matching* (FTM) algorithm. This section therefore introduces the FTM algorithm, as originally proposed by Kumar *et al.* [?]. We first describe the notations used throughout the rest of the article, and then describe the main steps of the algorithm.

2.4.1 FTM Notations and Overview

We define an ordered tree T as a directed graph $(N, \prec)$ where N is the non-empty set of nodes and $\prec$ a total order relation that can relate a *child* node $c \in N$ to its *parent* $p \in N$, as $c \prec p$, or *siblings*, as $s \in N$, as $c \prec s$.

In particular, we choose a total order rather than a partial one as the order of siblings has a strong semantic value for a webpage (e.g. the order of paragraphs).

In this article, we always consider *matchings* between two trees $T = (N, \prec)$ and $T' = (N', \prec')$.

Given two trees T and T', the FTM algorithm relies on the complete bipartite graph G between $N^* = N \cup \Theta$ and $N'^* = N' \cup \Theta'$, where Θ and Θ' are *no-match* nodes. The fact that G is complete means that every nodes of T^* shares exactly one edge with every nodes of T'^*. Formally, we thus have $E(G) = N^* \times N'^*$ where $E(G)$ are the edges of the graph G. An edge $e = (n, n') \in E(G)$ between $n \in N^*$ and $n' \in N'^*$ represents the matching of n with n'. Each edge linking a tuple (n, n') is called a *match*. So, intuitively, G represents all possible matchings between nodes of T^* and T'^* (cf. Figure **??**).

Formally, we call *matching* and note $M \subset E(G)$, a subset of edges selected from G. A matching M is said to be *full iff* each node in N has exactly one edge in M that links it to a node in N'^* and, inversely, each node in N' has exactly one edge

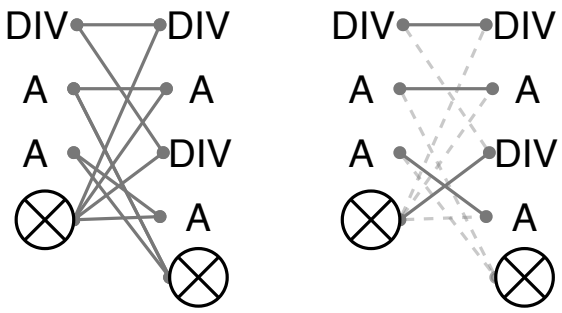


Figure 2.2: Building a bipartite graph G representing the set of all possible matchings (left) and then compute the optimal full matching (right).

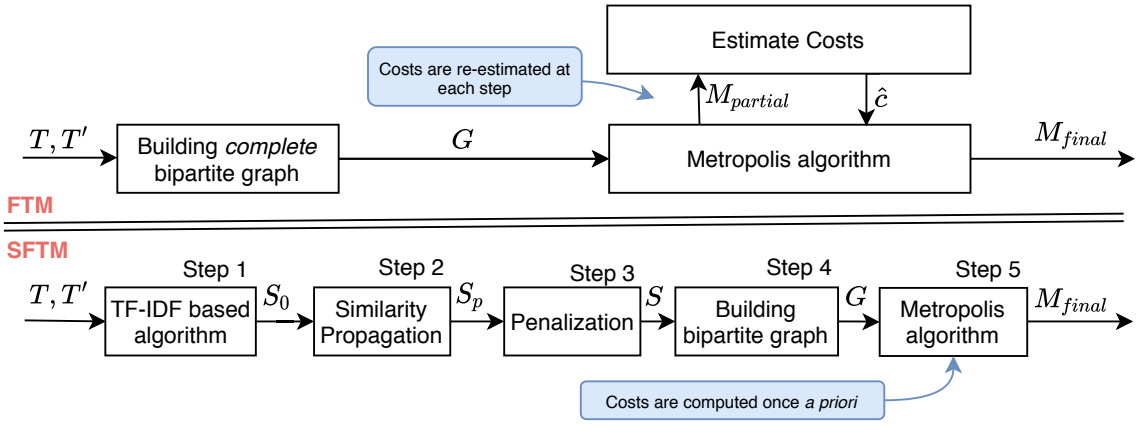


Figure 2.3: Steps to compute a full matching between two trees T and T'. Upper part covers FTM, while lower part is SFTM.

in M that links it to a node in N^*. Since matchings need to be *full*, the auxiliary *no-match* nodes Θ_1, Θ_2 are required to cope with insertion and deletion operations. The set of possible *full* matchings is restricted to the set of matchings satisfying that every node in $N \cup N'$ is covered by exactly one edge. *No-match* nodes are the only nodes allowed to be involved in multiple edges.

Given an edge $e = (n, n') \in E(G)$ linking n to n', FTM defines the cost $c(e)$ to quantify how different n and n' are, considering both their labels and the topology of the tree. Starting from the bipartite graph G describing all possible matchings, the idea behind FTM is to compute the costs $c(e)$ of each edge $e \in E(G)$ and to find the optimal matching with respect to these estimated costs—*i.e.*, to select the set of edges $M \subset E(G)$, such that M is *full* and $c(M)$ is minimal (where $c(M) = \sum_{e \in M} c(e)$).

The upper part of Figure **??** describes the main steps involved in computing the final full matching between T and T'.

2.4.2 Cost Estimation

As FTM provides a wide flexibility regarding possible matchings, the design of the cost function c is a key parameter in order to obtain a matching that takes into account both the labels and the topology of the trees. Typically, the cost $c(e)$ of an edge e between two nodes n and n' is estimated by FTM as follows:

$$c(e) = \begin{cases} w_n & \text{if } n \text{ or } n' \in \{\Theta, \Theta'\} \\ w_r c_r(e) + w_a c_a(e) + w_s c_s(e) & \text{otherwise} \end{cases} \tag{2.1}$$

where Θ, Θ' are *no-match* nodes, w_n is the penalty when failing to match one of the edge ends, $c_r(e)$, $c_a(e)$ and $c_s(e)$ are the cost of *relabeling*, *violating ancestry relationship* and *violating sibling group*, respectively, and w_r, w_r and w_r their associated weight in the cost function. w_n, w_r, c_r, w_a and w_s are parameters of the cost function that depend on the kind of matching the user requires. By extension, we note $c(M) = \sum_{e \in M} c(e)$ the cost of a matching M.

Given $e = (n, n')$, the ancestry and sibling costs, $c_a(e)$ and $c_s(e)$, model the changes in topology that matching n with n' entails. Unfortunately, we can only estimate the costs c_a and c_s if we have access to a full matching, as both costs require a knowledge on how other nodes in the tree were matched (*e.g.*, c_a involves counting the number of children of n matched with nodes that are not children of n'). In order to circumvent the problem, FTM rather considers the approximate costs $\hat{c_a}, \hat{c_s}$ that can be estimated from bounds on the different components of the cost c. Practically, in order to generate one possible full matching, FTM iteratively selects edges in G and, each time an edge is selected, the bounds of c are tightened (we can approximate c more precisely), which means that the costs $\hat{c_a}, \hat{c_s}$ keep being re-estimated along iterations (cf. upper part of Figure **??**). The need to re-estimate the approximated costs after each edge selection actually imposes some critical limitations on the scalability of the algorithm.

2.4.3 Metropolis Algorithm

Finding the optimal matching, given the graph G and the cost function c is a challenging problem, the authors even proved in [**?**] that this problem is NP-hard. Consequently, the authors described how to use the Metropolis algorithm [**?**] to approximate the optimal matching. The Metropolis algorithm provides a way to explore a probability distribution by random walking through samples. FTM uses this algo-

rithm to random walk through several full matchings, and select the least costly. The METROPOLIS algorithm requires to be configured with:

1. An initial sample (full matching) M_0,
2. A suggestion function (alternative matching) $M_t \mapsto M_{t+1}$,
3. An objective function to maximize: $f : M \mapsto$ quality of M,
4. The number of random walks before returning the best value.

Kumar *et al.* defines the objective function f by:

$$f_{FTM}(M) = \exp(-\beta\ c(M)) \tag{2.2}$$

In order to suggest a matching M_{t+1} from a previously accepted one M_t, FTM selects a random number of edges from M_t to keep, sorts remaining edges by increasing costs and iterate through the ordered edges with a probability γ to select it. Once an edge $e = (n, n')$ is selected, all edges connected to n and n' are removed from G, and approximate costs need to be re-estimated for all remaining edges, and then sorted so we can select another edge. The process is repeated until an alternative full matching is obtained. Therefore, despite using the METROPOLIS algorithm to reduce the time complexity of the problem, the overall algorithm remains prohibitively costly to compute (cf. Section **??**), notably due to the continuous re-estimation of the approximated costs at each step of the full matching generation.

2.4.4 Complexity Analysis

The original FTM article [**?**] does not report on the complexity nor the computation time of the algorithm. We, therefore, provide an analysis of FTM's theoretical complexity in order to compare it to the one of SFTM (cf. Section **??**).

When discussing complexity, to simplify the notations, we consider the matching of two trees with the same number of nodes and we note N the number of nodes of both trees.

Complete bipartite graph G Building the complete bipartite graph requires matching each node from T to one node from T', which requires $O(N^2)$ operations.

METROPOLIS algorithm For each iteration of the METROPOLIS algorithm, FTM has to suggest a new matching. In the worst case, the algorithm should choose among all N^2 edges. Each time an edge between e_1 and e_2 is selected, all other edges connected to e_1 and e_2 are pruned and remaining costs requires to be re-estimated.

It means that costs have to be re-estimated and sorted for N^2 edges, then $(N-1)^2$ edges (after selection and pruning) and so on, until all edges have been selected or pruned. This implies that the total number of times the costs are re-estimated and sorted is in $O(\sum_{n=0}^{N} n^2) = O(N^3)$. Estimating the cost for a given edge linking e_1 and e_2 involves counting the number of potential ancestry and sibling violations, which requires going through all edges connected to siblings and children of e_1 and e_2. Even if we assume the number of siblings and children is independent from N, it still means that estimating the cost of one edge requires $O(N)$ operations. Thus, in the worst case, the amount of operations required by FTM for each iteration of the METROPOLIS algorithm is in $O(\sum_{n=0}^{N} n^3) = O(N^4)$ (using Faulhaber's Formula).

Overall, the METROPOLIS step is the one with highest complexity, which means that the complexity of the FTM algorithm is in $O(N^4)$ where N is the number of nodes to match.

2.5 Similarity-based FTM (SFTM)

Based on the above complexity analysis, *Similarity-based Flexible Tree Matching* (SFTM) replaces the cost system of FTM by a similarity-based cost that can be computed once *a priori* (cf. Figure **??**). This approach drastically improves computation times and rather exposes a parameter that can be tuned to find the desired trade-off between computation time and matching accuracy (cf. Section **??**).

Given two trees $T = (N, \prec)$ and $T' = (N', \prec')$, SFTM relies on the specification of a *similarity metric* between nodes $n \in N$ and $n' \in N'$. We compute this similarity metric for all pairs of nodes (n, n') using *i)* inverted indices for labels and *ii)* label propagation and some penalization heuristics for the topology. We build a bipartite graph G between nodes of T and T' using this similarity metric to compute the costs and apply the Metropolis algorithm to approximate the optimal full matching from G. This new similarity measure allows SFTM to improve the FTM algorithm in two key aspects:

1. when building G, we do not create all $|N| \times |N'|$ possible edges. We only consider edges linking two nodes with a non-null similarity; and
2. when generating a full matching, costs do not need to be updated as these costs solely depend on our similarity measure.

In this section, we therefore (a) introduce our new similarity metric, and (b) describe how we leverage it to approximate the optimal full matching.

2.5.1 Overview of Similarity-based Matching

The similarity metric between nodes N and N' is computed in two main steps: 1. we compute s_0, the initial similarity function using only *labels* of the trees individually, and then 2. we transform s_0 to take into account the topology of the tree and compute our final similarity function s. The computation of s_0 leverages inverted index techniques traditionally used to query text in a large document database. In our case, the documents we query against are N, while queries are extracted from N'. Figure **??** illustrates the different steps described in this section.

Initial Similarity (step 1)

To compute the initial similarity s_0 between N and N' (cf. *step 1* in Figure **??**), we independently compare the labels of N and N' using the *Term Frequency-Inverse Document Frequency* (TF-IDF). The resulting initial node similarity s_0 does not take the topology of the trees into account.

In order to take into account relabeling cost between nodes, some existing solutions (*e.g.*, APTED) allow the user to input a pairwise comparison function $label(n), label(m) \mapsto$ *similarity score*. However, computing this similarity score for all the pairs of nodes requires $O(N^2)$ operations. Thus, to reduce the number of operations, SFTM uses—instead—inverted indices: given a tokenize function $tokenize : n \mapsto$ *token list*, SFTM 1. decomposes each node n from N into a set of tokens (as defined by the *tokenize* function), and then 2. iterates through tokens of nodes n' from N' to increase the value of $s_0(n, n')$ for each token n and n' have in common. Section **??** provides a detailed description of the function *tokenize* we use in our evaluation.

Decomposing nodes from N into tokens allows SFTM to build an inverted index TM (*Token Map*), which maps every token tk with the list of nodes of N that contains tk. The idea behind the inverted index TM is to use the information that a node $n \in N$ contains a token as a differentiating feature of n allowing to quickly compare it to nodes in N'. If a token tk appears in all nodes N, this token has no differentiating power. In general, the rarest a token, the more differentiating it is. This idea is very common in *Natural Language Processing* (NLP) and a common tool to measure how rare is a token in TF-IDF [**?**] and more precisely, the *Inverted Document Frequency* (IDF) part of the formula. Applying TF-IDF to our similarity yields the following

definition:

$$IDF(tk) = log(|N|/|TM[tk]|) \tag{2.3}$$

$$s_0(n, n') = \sum_{tk \in TK} IDF(tk) \tag{2.4}$$

Where $TK = tokens(n) \cap tokens(n')$. The function IDF is a measure of how rare a token is, $|TM[tk]|$ is the number of nodes containing the token tk and *tokens* refers to the user input tokenize function. Intuitively, we retrieve the tokens shared between nodes n and n' and, for each common token tk, we increase $s(n, n')$ by a high value if tk is rare and a low value if tk is common. In Section **??**, we provide a detailed implementation of how to compute the initial similarity s_0.

Tokens that appear in many nodes have little impact on the final score—*i.e.*, low IDF—yet have a very negative impact on the computation time. In our algorithm, we expose the sublinear threshold function $f : N \mapsto f(N) < N$ as a parameter of the algorithm. We use f to filter out all the tokens appearing in more than $f(N)$ nodes. Therefore, f provides a balance between computation time and matching quality: when $N - f(N)$ decreases, computation times and matching quality increase. In Section **??**, we discuss how $f(N)$ influences the worst-case theoretical complexity.

Local Topology (step 2)

s_0 represents the similarity between node labels, but does not take into account the topology of the trees. To weight in local topology similarities, we propagate the score of each node pair to their offspring and siblings. This idea of propagation is inspired by recent *Graph Convolutional Network* (GCN) techniques [**?**].

The original FTM algorithm includes two terms in the cost function, c_a (ancestry cost) and c_s (sibling cost), which reflect the topology of the trees. Since we do not use these terms (as they require too much computation time), we need our similarity score to reflect both the similarity of node labels and the similarity of the local topology. Therefore, we first compute the score matrix s_0, based on the label similarity we described above, and then we update this score to take into account the matching score of the parents of n and n'. By doing so, n has a higher similarity score with n' if their respective parents or children are also similar.

Beginning at s_0, at each step i and for all pairs that have a non-null initial score $\{(n, n') \in N \times N' | s_0 \neq 0\}$, we first compute:

$$s_i(n, n') \leftarrow s_{i-1}(n, n') + w_i \times s_{i-1}(p(n), p(n')) \tag{2.5}$$

where $p(n) \in N$ refers to the parent of node n.

Similarly, we then increase the score of the parents of n, n':

$$s_i(p(n), p(m)) \leftarrow s_{i-1}(p(n), p(n')) + v_i \times s_{i-1}(n, n') \tag{2.6}$$

where $w_0, w_1 \dots w_P$ and $v_0, v_1 ... v_P$ are topology weights. We repeat the process P times (P for propagation) where P is a parameter of SFTM. The resulting function s_P then reflects both label similarity and local topology similarity.

Intuitively, at each iteration, we propagate information further up in the tree. This is why, the weight sequences w and v should be decreasing so that close kinship among nodes prevails. From our experiments, we advice the following values for the $P = 3$ weights: $w_0 = 0.4, w_1 = 0.04, w_2 = 0.004$ and $w_0 = 0.8, w_1 = 0.08, w_2 = 0.008$. These values were used and unchanged for all results presented in the empirical evaluation **??**, leading to high accuracy on a large variety of web documents.

Penalization (step 3)

There are two main drawbacks to the way we propagate the scores in step 2: 1. the scores are still almost exclusively based on labels, 2. nodes with many children may get an unfair score boost from the propagation.

While (2) can be fixed by normalizing the propagation according to the number of children, the normalization would also potentially remove valuable information. Instead, for each pair (n, n'), we rather apply a penalization proportional to the difference between the number of children of n and n':

$$s(n, n') = s_P \times (1 - penalty(n, n')) \tag{2.7}$$

where $penalty(n, n') \mapsto [0, 1]$ is the children penalization defined by:

$$penalty(n, n') = \frac{|(|children(n)| - |children(n')|)|}{max(|children(n)|, |children(n')|)} \tag{2.8}$$

where $|ch(n)|$ is the number of children nodes of n. This step yields the final score function s, defined for each couple (n, n').

Building the bipartite graph G (step 4)

Using our final score function s, we can now build the bipartite graph G: we iterate over all nodes $n \in N$ and we create an edge $e = (n, n')$ for each pair of nodes such

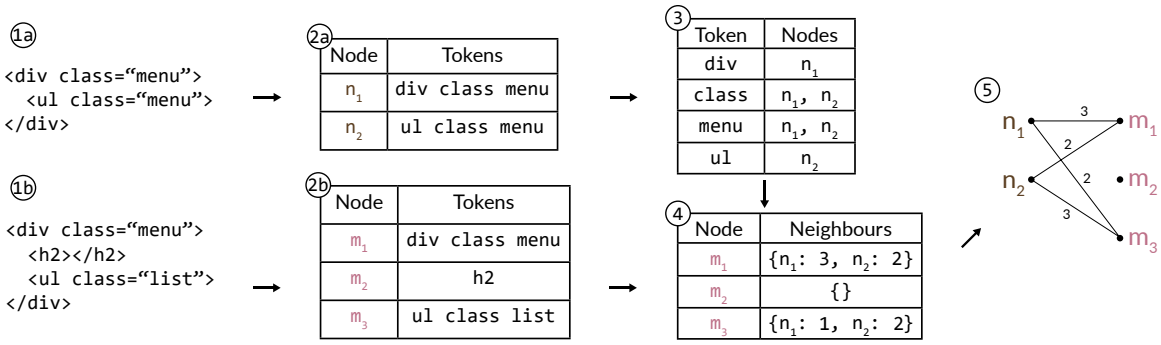


Figure 2.4: Creating the bipartite graph G from two example DOMs T, T'. (1a,b) are the input DOMs, (2a,b) the extracted tokens, (3) the inverted index TM, (4) the neighbours dictionaries, and (5) the resulting bipartite graph G. For simplicity, the figure shows a matching where $IDF(tk) = 1$, $P = 0$ and no-match nodes are not displayed.

that $s_P(n, n') \neq 0$ and associate it with the cost $c(n, n') = 1/(1 + s_P(n, n'))$. Our resulting cost function is thus defined as follows:

$$c_{SFTM}(e) = \begin{cases} w_n, & \text{if } n \text{ or } n' \text{ is a no-match node} \\ \frac{1}{1+s_p(n,n')}, & \text{otherwise} \end{cases} \tag{2.9}$$

Importantly, unlike the bipartite graph built in the FTM algorithm, the resulting bipartite graph G_{SFTM} is *not complete* as only edges, such that $s_p(n, n') \neq 0$ are considered. This is one of the key differences allowing SFTM to drastically improve computation times.

2.5.2 Implementation Details

In the previous section, we introduced the SFTM algorithm and described how it compares to FTM. In this section, we describe more precisely how we implement the different steps of SFTM.

Node Similarity (step 1, 2 and 3)

Let us consider two trees T and T'. We first build the dictionary TM, an inverted index—*i.e.*, each entry of TM is a tuple $(token, nodes)$ where $token$ is a token (usually a string) and $nodes$ is a *set* of all $n \in N$ that contains $token$. Figure **??** (2a,b) depicts two examples of inverted index. We note $TMmap[key]$ the set of $nodes$ whose key in TM is key. In Section **??**, we further describe how we sort HTML nodes into tokens.

Given the inverted index TM, we define the function $IDF : tk \mapsto log(|N|/|TM[tk]|)$. In order to limit the complexity of our algorithm, we remove every token $tk \in TM$ that is contained by more than $f(N) = \sqrt{N}$ nodes, where f is the chosen sub-linear threshold function. This is equivalent to putting a threshold on IDF to only keep tokens $\{tk \in TM | IDF(tk) > log(\sqrt{N})\}$. Removing the most common tokens has a limited impact on matching quality, since these are exactly the tokens that provide the least information on the nodes they appear in.

Input:
n': a node in N
TM: token map, dictionary of nodes from T per token
Result: neighbors: a dictionary of scores per node in T
neighbors $\leftarrow$ *new Dictionary*()
foreach *tk in tokens(n')* **do**
  foreach n *in* $TM[tk]$ **do**
    $neighbors[n] += IDF(tk)$
  end
end
return neighbors

Algorithm 1: For a given node $n' \in N'$, compute similarity score $s_0(n, n')$ with all $n \in N$, such that $s_0 > 0$

Once we have the token index TM and the function IDF, we apply Algorithm **??** on each node $n' \in N'$. In Algorithm **??**, we first compute the tokens of node n' and, for each token tk, we use TM to retrieve the nodes $n \in N$ that contain the token tk. Each node n thus retrieved is considered as a *neighbor* of n'—*i.e.*, $s_0(n, n') \neq 0$. Finally, for each neighbour n of n', we add $IDF(tk)$ to the current score $s_0(n, n')$. At this point, we have a $neighbors(n')$ dictionary for each node $n' \in N'$. Each $neighbors(n')$ dictionary contains all non-null matching scores: $\forall n \in keys(neighbors(n')), neighbors(n')[n] = s_0(n, n')$. Using the Equation **??**, we can now easily compute s_p and s.

Building the Token Vector

The actual labels are never directly used by SFTM. The algorithm only leverages the tokens extracted from these labels. The way we choose to extract the tokens contained in a node n thus strongly influences the quality of our similarity score. We implemented the following function *tokens* to report all the tokens of a node n. Given n, an HTML node representing a `tag`:

```
<tag att_1="val_1" ... att_2="val_2">
```

```
    CONTENT
</tag>
```

where l is the number of attributes, $(att_i, val_i), i \in [1, l]$ are the attribute/value pairs of n and the absolute XPath of n is $xPath(n)$. We decompose n into the following tokens:

$$tokens(n) = \{xPath(n), \texttt{tag}, att_1..a_l, tok(val_1)..tok(val_l)\} \tag{2.10}$$

where *tok* is a standard string tokenizing function that takes a string and splits it into a list of tokens on each non Latin character. The absolute XPath of a node n in a tree is the full path from the root to the element where ranks of the nodes are indicated when necessary—*e.g.*, `html/body/div[2]/p`.

SFTM does not include the text content of the nodes in the extracted token vectors. This decision allow to match pages in different languages or containing different content (e.g. news website) in a robust way.

Building G (step 4)

Using Equation **??**, we compute the cost $c(n, n')$ for each couple (n, n') where $s_p(n, n') \neq 0$. Then, for each node $n' \in N'$, we add one edge for all nodes $values(neighbours(n')) \subset N$.

Metropolis Algorithm (step 5)

Once we built the graph G with its associated costs, we need to find the set of edges M in G that represents the best full matching between T and T'. In order to do so, we apply the METROPOLIS algorithm in a different way than FTM does: 1. we adopt an alternative objective function, and 2. SFTM matching suggestion function is faster to compute, as costs never need to be re-estimated.

Typically, FTM uses the objective function $f_{FTM}(M) = \exp(-\beta\ c(M))$. In the original FTM article, the authors noted that the parameter β seemed to depend on $|M|$. In order to avoid this dependency, we therefore normalize the total cost:

$$f_{SFTM}(M) = \exp(-\beta\ \frac{c(M)}{|M|}) \tag{2.11}$$

The function $suggestMatching : M_i \mapsto M_{i+1}$ takes a full matching M_i and returns a full matching M_{i+1} related to M_i. In Algorithm **??**,

1. $selectEdgeFrom(edges)$ loops through *edges* (in order) and, at each iteration j, has a probability $\gamma \in [0, 1]$ to stop and return $edges[j]$,

2. *connectedEdges*(*edge*), where *edge* connects u and v, returns the set E of all edges connected to u or v (note that $edge \in E$).

Data: G : The bipartite graph
Input: M_i: A full matching
Result: M_{i+1}: the suggested full matching
$M_{i+1} \leftarrow []$
$remainingEdges \leftarrow sortedEdges(g)$
$toKeep \leftarrow randomInt(0, |M_i|)$
for $j = 0 \ldots toKeep$ **do**
    edge ← remainingEdges.first
    M_{i+1}.add(edge)
    remainingEdges.removeAll(connectedEdges(edge))
end
while *remainingEdges is not empty* **do**
    $edge \leftarrow selectEdgeFrom(remainingEdges)$
    M_{i+1}.add(edge)
    remainingEdges.removeAll(connectedEdges(edge))
end
return M_{t+1}

Algorithm 2: Suggest a new matching

In practice, we first compute all the connected nodes and edges before storing them as dictionaries, so that the function *connectedEdges* in Algorithm **??** can be computed in $O(1)$ time. It is worth noting that, to allow fast removal, the list *remainingEdges* is implemented as a double-linked list. The parameter γ defines a trade-off between exploration (low γ) and exploitation (high γ). For the Metropolis related parameters, we used mostly the values advised in the original FTM article [**?**]: $\gamma = 0.8, \beta = 2.5$ and a number of iterations of 10.

2.5.3 Complexity Analysis

We are interested in evaluating the time complexity of the algorithm with respect to the size of both trees N. In our analysis, we consider that $N_{tk} = max(|tokens(n)|, n \in nodes(T))$, the maximum number of tokens per node is a constant since it does not evolve with N.

When building G, we first compute the inverted index TM, which requires to iterate through the tokens of all the nodes in T, and thus implies a complexity in $O(N \cdot N_{tk}) = O(N)$.

To find the neighbours of nodes from T' using TM, we iterate through all the nodes in T', while each node in T' has N_{tk} tokens. The number of nodes containing

a token is artificially limited to $f(N)$. Thus, building the similarity function s_0 takes $O(N \cdot f(N))$ time.

For each node n' in T', we create an edge for each neighbor n of T. Each token $tk \in tokens(n')$ adds up to $f(N)$ neighbors. It means that the total number of edges is in $O(N \cdot N_{tk} \cdot f(N)) = O(N \cdot f(N))$.

Before executing the METROPOLIS algorithm on G, we sort all the edges by cost, which takes $O(N \cdot f(N) \cdot log(N \cdot f(N))) = O(N \cdot f(N) \cdot log(N))$ (as $f(N) \leq N$). Finally, at each step of the METROPOLIS algorithm, we run the *suggestMatching* function, which prunes a maximum of $O(f(N))$ neighbors for each one of the N edges it selects.

Overall, sorting all edges requires the highest theoretical complexity: $O(N \cdot f(N) \cdot log(N))$. If no threshold is set—*i.e.*, $f(N) = N$—then the worst-case overall complexity of SFTM is $O(N^2 \cdot log(N))$, which keeps outperforming TED ($O(N^3)$) and FTM ($O(N^4)$).

In this evaluation, we used $f(N) = \sqrt{N}$, which leads to a theoretical worst-case complexity in $O(N \cdot \sqrt{N} \cdot log(N))$.

2.6 Empirical Evaluation

The objective of this evaluation is to assess that:

1. the quality of the matchings reported by SFTM compares with the baselines we selected—APTED and XYDIFF—and
2. SFTM offers practical computation times on real-life web pages.

2.6.1 Input Web Document Dataset

We need to assess the ability of SFTM to match the nodes between two slightly different web pages d and d'. To measure and compare the accuracy of all studied solutions, we must have access to the ground truth matching between d and d'—*i.e.*, for each node n in d, what is the true matching node n' in d'.

To the best of our knowledge, there is no established and public benchmark that include such pairs of trees, along with the ground truth matching of their nodes. Creating such a benchmark is challenging. Existing matching solutions usually do not provide any qualitative empirical benchmark [?, ?, ?, ?, ?, ?] and challenging matching problems involve thousands of nodes, which makes manual labeling error prone for humans (both trees could not even be rendered on the same screen). Therefore, we

built a a semi-synthetic dataset built from mutations applied to real-life web pages, thus obtaining a large-scale dataset in which the ground truth is known.

DOM mutation To build a grounded dataset of (d, d') pairs—*i.e.*, where the ground truth (perfect matching) is known—we developed a mutation-based tool that operates as follows:

1. we construct the DOM d from an input web document,
2. for each element of d, we generate a unique signature attribute,
3. for each original DOM d, we randomly generate a set of mutated versions: the *mutants*. Each *mutant* d' is stored along with the precisely described set of mutations that was applied to d to obtain d'. Importantly, the signature tags of the elements in d are transferred to d', which constitutes the perfect matching between d and d'. These signatures are obviously ignored when applying the matching algorithms.

In our tool, most attention has been dedicated to the choice of relevant mutations to apply. We therefore relied on the expertise of web developers to identify the most common changes that can apply to DOM. Their feedback led to the identification of the following list of mutation operations:

Category	Mutation Operators
Structure	`Remove, duplicate, wrap, unwrap, swap`
Attribute	`Remove, remove words`
Content	`Replace, remove, remove words, change letters`

where *Structure: remove* removes an element and its children (recursively), *duplicate* duplicates a subtree, applies *mutate* to duplicated subtree and insert the subtree anywhere in the tree, *wrap* wraps an element within a new *div* container, *unwrap* removes an element e and attach the children of e to the parent of e, *swap* swaps the position of two sibling elements, *Attribute: remove* removes an attribute with its value, *Attribute: remove words* removes a random number of tokens from the value of an attribute, *Content: replace* replaces the content of an element with a random text whose size is close to the original, *Content: change letters* replaces a few letters in the content of an element, *Content: remove* removes the content of an element, *Content: remove words* removes random tokens from the content of an element.

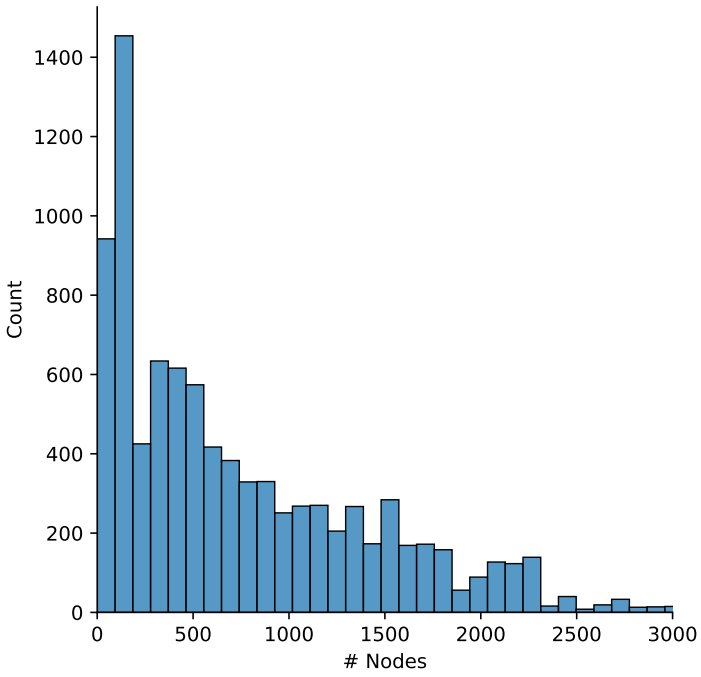


Figure 2.5: Distribution of DOM sizes (in terms of nodes) in the dataset.

We believe that the above mutations are representative of a wide range of changes that apply in web pages, although they may not perfectly cover all the cases encountered in practice. In particular, the distribution of these mutations might not be uniform in real life—*i.e.* some mutations might happen more than others. Yet, this evaluation intends to compare the sensitivity of studied matching algorithms to common mutations, which remains a relevant context to estimate and compare their quality.

Input document sample We fed our mutation tool with the home pages of the Top 1K Alexa websites.[2] Alexa provides a list of websites ordered by popularity, thus providing a representative list of web pages of variable complexities. For each original DOM d, we created 10 mutants $d'_0 \dots d'_9$ with a ratio of mutated nodes ranging from 0 to 50% of the total number of nodes on the page, $|nodes(d)|$.

Overall, we obtained a dataset composed of $6,276$ document pairs d, d'_n that could be correctly processed by the algorithms under study. Figure **??** reports on the size distribution, in number of nodes, of original and mutated web documents included in this dataset.

[2]https://www.alexa.com/topsites

2.6.2 Baseline algorithms

Given no implementation of the original FTM algorithm is available, we implemented and evaluated it, but the computation times and space complexity of this implementation were too high to run the algorithm on real-life web documents (*e.g.* for a tree of 58 nodes, the computation took 1 hour).

We thus mainly compare SFTM to APTED and XyDiff. APTED is the reference implementation of TED that reports on the best performance so far. The implementation of APTED used for this evaluation is the one provided by the authors of [**?**, **?**]. Since APTED yields the optimal solution to the TED problem, TED is theoretically superior in accuracy to all more restricted solutions (see Section **??**).

XyDiff is the most widely-known and used algorithm to efficiently match XML documents. Unlike APTED, XyDiff does not return an optimal result, it instead focuses on speed which makes it a complementary candidate to APTED as a baseline. In order to use XyDiff on HTML pages we had to convert the HTML into XHTML, which mostly means closing unclosed tags (*e.g.*, input tags). We used an existing open source implementation of XyDiff.[3] We consider the pairs (d, d') taken from the above input dataset, and we systematically ran SFTM, APTED, and XyDiff algorithms with each pair to match d with d' on the same machine.

Ground truth When building the dataset, we keep track of nodes' signature so that we always know which nodes from d should match with nodes from d'. This ground truth is hidden from the evaluated algorithms, but is used *a posteriori* to measure and compare the quality of the matchings computed by the algorithms under evaluation.

2.6.3 Experimental Results

All the results in this section have been obtained by running all three algorithms on the same server containing 252 GB of RAM and an Intel(R) Xeon(R) CPU E5-2660 v3 @ 2.60 GHz.

Matching quality The signature tags injected in nodes from d and d' allow us to assess the quality of the matchings by comparing to the ideal matching M_{ideal}. For the qualitative analysis, we model the tree matching algorithm as a binary classification problem: Given two trees T and T' containing the set of nodes N and N' respectively, $N \times N'$ is the set of all possible matches. We consider the matching $M_a \subset N \times N'$

[3] https://github.com/fdintino/xydiff

produced by a tree matching solution a. Then, a match $e = (n, n') \in M$ is classified as positive by a if $e \in M_a$. All matches that should be positive are in the ideal matching M_{ideal}. All possibilities are summarized in the following confusion matrix:

	$e \in M_{ideal}$	$e \notin M_{ideal}$
$e \in M_a$	**True Positive**	False Positive
$e \notin M_a$	False Positive	**True Negative**

Using the above confusion matrix, we can compute the *precision*, *recall*, metrics and the *F1 score*, which are very commonly considered for binary classification problems.

Figure **??** reports on the precision, recall and F1 score of the 3 tree matching solutions we compared, namely SFTM, APTED, and XyDiff. As expected, the accuracy of all solutions decreases when the mutation ratio increases. However, for all the reported metrics, SFTM clearly outperforms both XyDiff and APTED. For both APTED and XyDiff, we believe the lack of accuracy stems from the lack of flexibility when matching labels. XyDiff relies entirely on hashing subtrees of the document and match subtrees with identical hash. While this approach might be robust to small structural mutations, it is naturally very sensitive to large amounts of mutations when both structures and labels are mutated. Similarly, TED compares the labels of most pairs of nodes and generate an associated cost of `1` when the labels are identical and `0` when they are different (no gradual costs if the labels are similar).

Completion time For each couple (d, d') retrieved from the dataset, we measured the time taken by SFTM, APTED, and XyDiff to compute a full matching. Figure **??** reports on the average time to match DOM couples of increasing size (in terms of number of nodes) for all three solutions. XyDiff exhibits very fast computation speed and despite its numerous optimizations, APTED's computation times increases exponentially large web documents. SFTM is not as fast as XyDiff, but seems to show reasonable growth when the size of web documents increases. Interestingly, APTED computation time varies greatly, which is due to the multiple heuristics used by this implementation to optimize the computation in certain situations.

Overall, one can observe that SFTM offers an interesting trade-off between two classes of tree matching algorithms: the ones maximizing accuracy at the cost of time, like APTED, and those minimizing the completion time at the cost of reduced accuracy, like XyDiff.

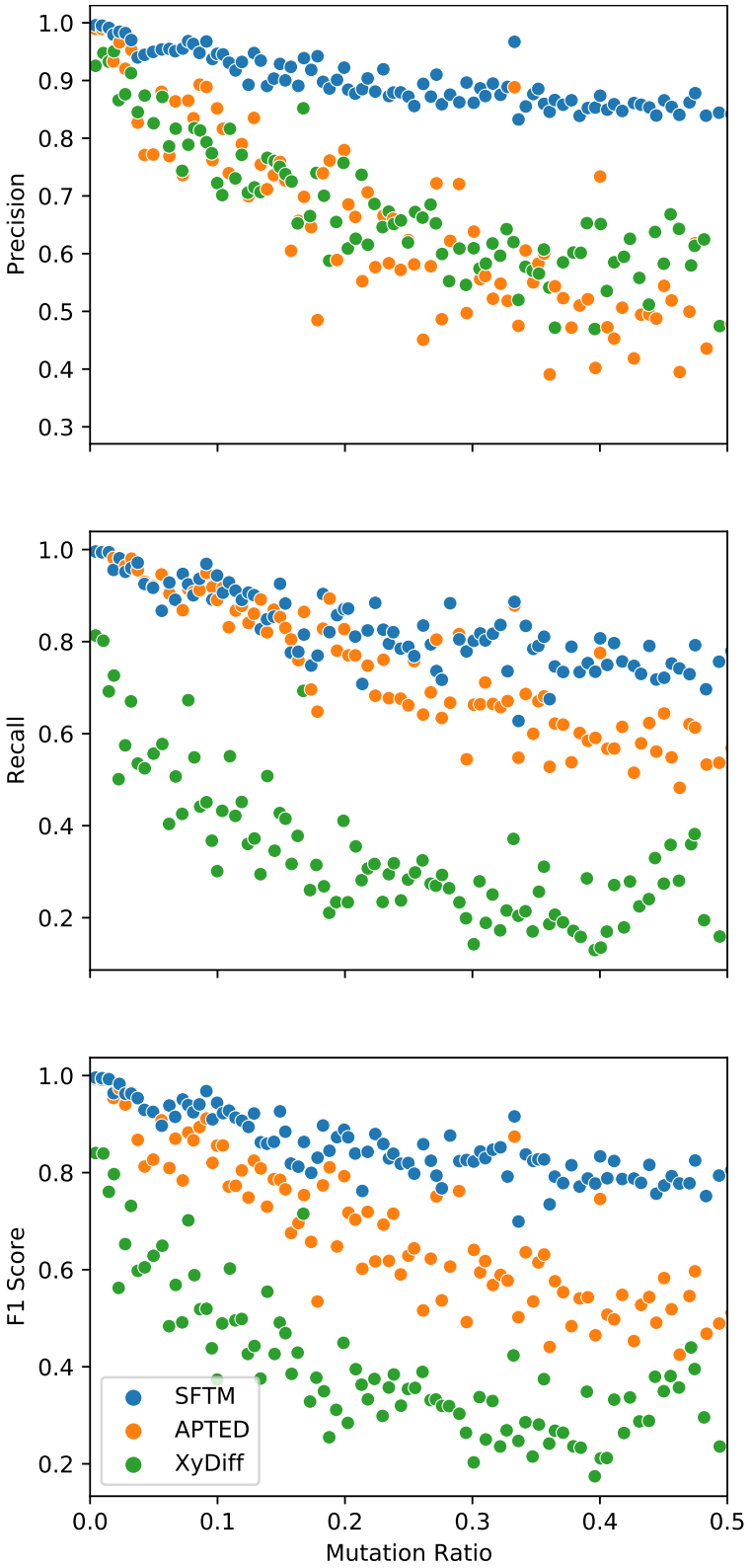


Figure 2.6: Precision, Recall, and F1 Score of SFTM, APTED, and XyDiff.

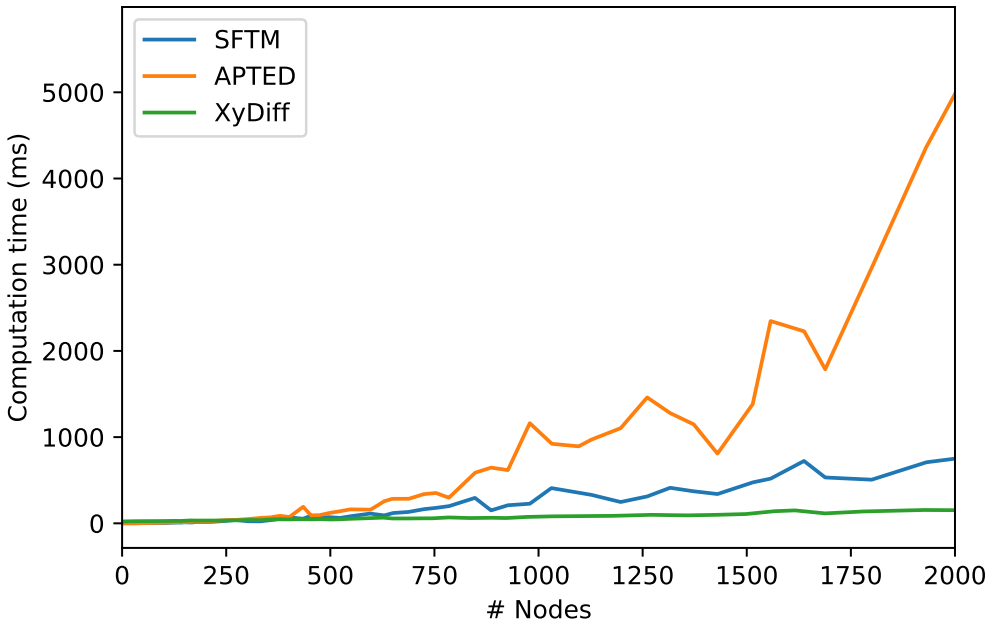


Figure 2.7: Computation times when matching trees of different sizes.

Matching efficiency The matching efficiency measures how fast a given solution can produce accurate results. The efficiency is a way to combine both accuracy and speed metrics into one that can be used to compare all solutions. In our case, we already showed that SFTM outperforms APTED in both accuracy and computation time. This efficiency measure is thus particularly interesting to compare SFTM to XyDiff, as SFTM outperforms XyDiff in terms of accuracy, but remains slower when it comes to speed. To compute this matching efficiency, we consider the same metric as [**?**]—*i.e.*, the number of good matches produced per millisecond. Figure **??** reports on the matching efficiency of all three matching solutions. One can observe that SFTM produces 7.7 good matches per millisecond on average, which is far above APTED and XyDiff that produce 3.6 and 2.4 good matches per millisecond, respectively.

Parameter sensitivity Since we aim at improving the performances of FTM in term of computation times, we study the sensitivity of the sub-linear threshold function f, which is a parameter that directly influences the computation time of the algorithm.

Figure **??**, therefore, reports on the evolution of SFTM performances when f varies. To study the sensitivity of f, we choose to use the power function $f(N) = N^{\alpha}$ as a threshold and display how the computation times and matching accuracy evolve with α.

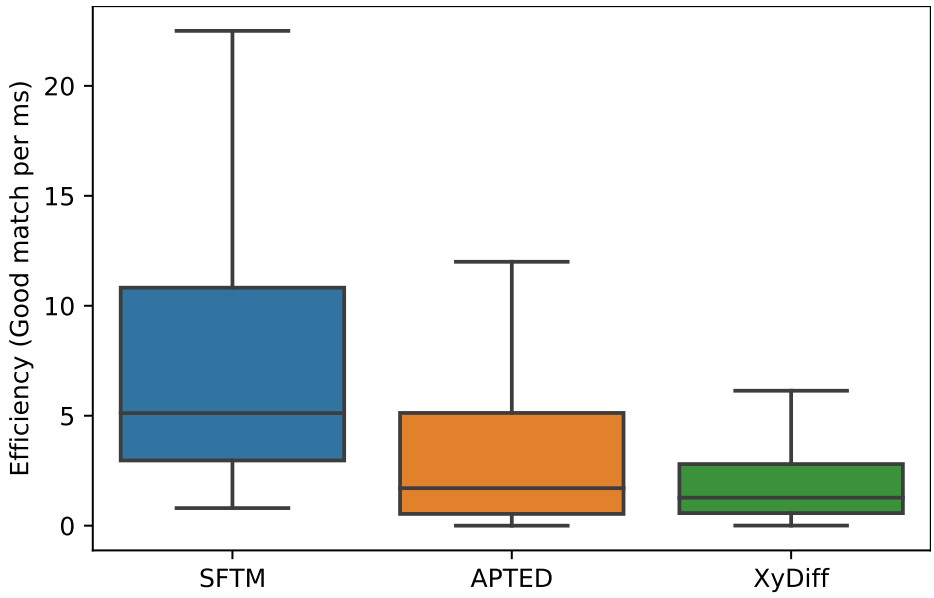


Figure 2.8: Matching efficiency of SFTM, APTED, and XyDiff.

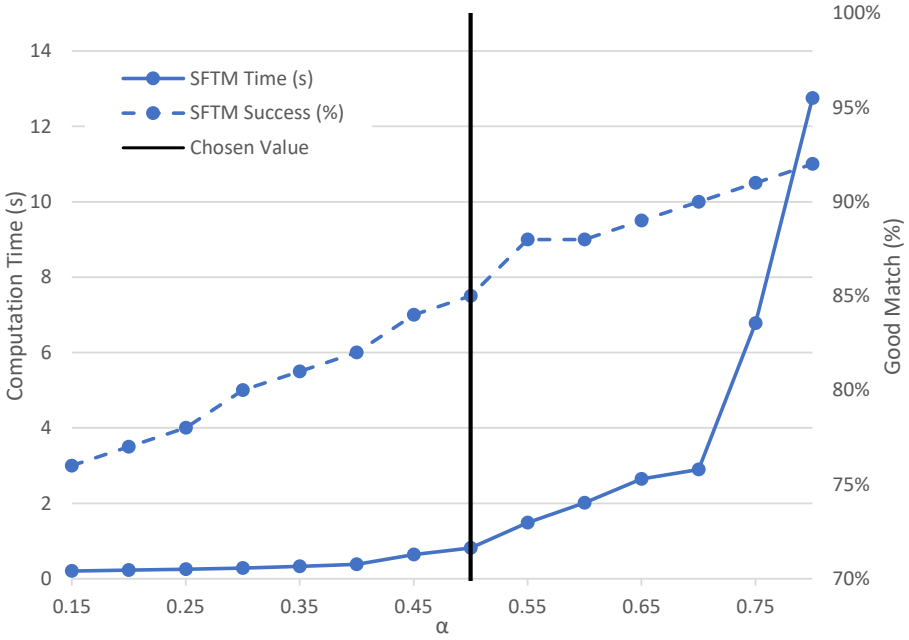


Figure 2.9: Performance of SFTM given $f(N) = N^{\alpha}$ according to α.

For this experiment, as we are interested in studying the sensitivity of the parameter α on the performances of SFTM, we therefore consider a subset of 243 pairs from the complete dataset used in previous sections (cf. Section **??**), which represents a 6% error margin with 95% confidence.

As expected, when α increases, the quality of the matching and the computation times increase. However, beyond a certain value of α, the increase of computation time is superior to the gain in accuracy: increasing α from 0.5 to 0.8 entails more than 10 times longer computation times for 8% gain in accuracy. Intuitively, this is because tokens contained in most nodes provide few relevant information (low IDF), but increase the complexity quadratically. In this article, we thus adopted $\alpha = 0.5$ (*i.e.*, $f(N) = \sqrt{N}$) as this value achieves good enough performances to demonstrate that SFTM can match two real-life web pages in practical time, with a minimum of compromise on quality.

2.7 Threats to Validity

The absolute values of completion times depend on the machine on which the algorithms were executed. As computations took time, we had to run both SFTM, APTED and XyDiff on a server, which is shared among several users. Although we paid a careful attention to isolate our benchmarks, the available resources of the server might have varied along execution thus impacting our results.

Our dataset contains the homepages of the Top 1k Alexa websites. The fact that our qualitative evaluation has only been conducted on homepages might have biased the results, as such pages might not be fully representative of the complexity of online documents. Yet, one can observe that the distribution of page sizes in our datasets offers a good diversity of situations.

The parameters used for SFTM and, in particular, the weights for the propagation may not be optimal. However, our evaluation shows that the adopted values succeed to report tree matchings that compete with the state-of-the-art accuracy in reasonable times and on a very large variety of web pages, which means the values we provided for the parameters do not require to be tuned for most web pages matching cases.

2.8 Conclusion & Perspectives

Comparing modern real-life web pages is a challenge for which traditional *Tree Edit Distance* (TED) and XyDiff solutions are too restricted and computationally expen-

sive. [?] introduced *Flexible Tree Matching* (FTM) to offer a restriction-free matching, but at the cost of prohibitive computational times.

This article thus introduced *Similarity-based Flexible Tree Matching* (SFTM), the first implementation of an advanced *Flexible Tree Matching* (FTM) algorithm with scalable computation times. We evaluated our solution using mutations on real-life web pages and we showed that SFTM outperforms XyDiff qualitatively and compares to TED, while significantly improving the computation time of the latter. Our proof of concept demonstrates that accurate matching of real-life web pages in practical time is possible.

Our *label-centric* approach to matching is significantly different than previous *structure-centric* techniques. In addition to providing a competitive solution to match web pages, we hope that our solution will encourage the development of solutions based on similar approaches. We also believe that having a robust algorithm to efficiently compare web pages will open up new perspectives within the web community.

In future work, we will further investigate how to improve the quality of the tree matchings by analyzing which situations cause SFTM to report mismatches and to establish guidelines to adjust the exposed parameters.

Finally, whether our work might be applicable to other trees than web DOMs remains to be demonstrated. Indeed, SFTM strongly relies on the fact that node labels in DOMs are highly differentiating (many specific attributes on each element), which is not the case for all kinds of trees.

Chapter 3

Erratum

3.1 abstract

Web applications are constantly evolving to integrate new features and fix reported bugs. Even an imperceptible change can sometimes entail significant modifications of the *Document Object Model* (DOM), which is the underlying model used by browsers to render all the elements included in a web application. Scripts that interact with web applications (*e.g.* web test scripts, crawlers, or robotic process automation) rely on this continuously evolving DOM which means they are often particularly fragile. More precisely, the major cause of breakages observed in automation scripts are *element locators*, which are identifiers used by automation scripts to navigate across the DOM. When the DOM evolves, these identifiers tend to break, thus causing the related scripts to no longer locate the intended target elements.

For this reason, several contributions explored the idea of automatically repairing broken locators on a page. These works attempt to repair a given broken locator by scanning all elements in the new DOM to find the most similar one. Unfortunately, this approach fails to scale when the complexity of web pages grows, leading either to long computation times or incorrect element repairs. This article, therefore, adopts a different perspective on this problem by introducing a new locator repair solution that leverages tree matching algorithms to relocate broken locators. This solution, named Erratum, implements a holistic approach to reduce the element search space, which greatly eases the locator repair task and drastically improves repair accuracy. We compare the robustness of Erratum on a large-scale benchmark composed of realistic and synthetic mutations applied to popular web applications currently deployed in production. Our empirical results demonstrate that Erratum outperforms the accuracy of WATER, a state-of-the-art solution, by 67%.

3.2 Introduction

The implementation of automated tasks on web applications (apps), like crawling or testing, often requires software engineers to locate specific elements in the DOM (*Document Object Model*) of a web page. To do so, software engineers or automation/testing tools often rely on CSS (*Cascading Style Sheets*) or XPath selectors to query the target elements they need to interact with. Unfortunately, such statically-defined locators tend to break along time and deployments of new versions of a web application. This often results in the failure of all the associated automation scripts (including test cases) that apply to the modified web pages.

While several existing works focus on repairing tests on GUI applications, there are surprisingly very few test repair solutions targeting web interfaces [?]. These solutions either propose to *i)* generate locators that are robust to changes (so-called *robust locator problem*), or *ii)* repair locators that are broken by the changes applied to the web pages (so-called *locator repair problem*). Unfortunately, most of the existing solutions in the literature fail to accurately relocate a broken locator, thus leaving all the related web automation scripts as broken [?]. More specifically, state-of-the-art solutions to the locator repair problem, WATER [?] and VISTA [?], tend to rely on the intrinsic properties of the element whose locator needs repairing to locate its matching element on the modified page. However, this approach fails to leverage the element position and relations with the rest of the DOM, thus ignoring valuable contextual insights that may greatly help to repair the locator.

In this article, we adopt a more holistic approach to the locator repair problem: instead of focusing on the element whose locator is broken individually, we leverage a tree matching algorithm to match all elements between the two DOM versions. Intuitively, using a holistic approach to repair a broken locator should significantly improve accuracy by reducing the search space of candidate elements in the new version of the page: for example, if the parent of the element whose locator is broken is easily identifiable (*e.g.*, the item of a menu) a tree matching algorithm will use this information to relocate the target element in the modified page with better accuracy. Additionally, if more than one locator is broken on a given web page, our approach will repair all of them consistently at once. The holistic solution we propose, named ERRATUM,[1] more specifically leverages an efficient *Similarity-based Flexible Tree Matching* (SFTM) algorithm to repair all broken locators by matching all changes in a web page with high accuracy, compared to the state-of-the-art solu-

[1] ERRATUM stands for "*rEpaiRing bRoken locATors Using tree Matching*"

tions. SFTM is a tree matching algorithm providing fast computation times and high accuracy when compared to other generic tree matching solutions. To do so, SFTM builds on a distinctive characteristic of DOM trees: the labels of the nodes (*i.e.*, node attributes and tags) contain a high amount of information that can be leveraged to prune the space of possible matchings between two trees.

Evaluating solutions to both robust locator and locator repair problems requires to build a dataset of web page versions—*i.e.*, (original page, modified page) pairs. Unfortunately, previous works assessed their contributions on hardly-reproducible benchmarks of limited sizes (never beyond a dozen of websites). In this article, we rather evaluate the robustness of our approach against the state of the art by introducing an open benchmark, which covers a wider range of changes that can be found in modern web apps. Concretely, our open benchmark considers over 83k+ locators on more than 650 web apps. It combines *i)* a *synthetic dataset* generated from random mutations applied to popular web apps and *ii)* a *realistic dataset* replaying real mutations observed in web apps from the Alexa Top 1K,[2] which ranks the most popular websites worldwide.

When evaluated on both datasets, our results demonstrate that ERRATUM outperforms the state-of-the-art solution, namely WATER [**?**], both in accuracy (67% improvement on average) and performances, when more than 3 locators require to be repaired in a web page.

Concerning the potential applications of ERRATUM, while we introduce and evaluate our solution within the well-studied context of locator repair, we also discuss a novel resilient architecture centered around ERRATUM allowing to entirely replace all locator-based interactions. This architecture intends to support much more interactive and robust script editions in the context of web testing, web crawling, and robotic process automation.

Summary Overall, the key contributions of this article consist of:

1. proposing a solution to the locator repair problem leveraging the principles of *Flexible Tree Matching* (FTM),
2. implementing and integrating an efficient algorithm, named ERRATUM, to repair broken locators,
3. providing a novel, reproducible, large-scale benchmark dataset to evaluate both the robust locator and locator repair problems,
4. reporting on an empirical evaluation of our approach when solving the locator

[2] `https://www.alexa.com/topsites`

repair problem,

5. proposing a novel script edition architecture centered on ERRATUM.

Outline The remainder of this article is organized as follows. Section ?? introduces the state-of-the-art approaches in the domain of robust locators and locator repair, before highlighting their shortcomings. Section ?? formalizes the locator problem we address in this article. Section ?? introduces our approach, ERRATUM, which leverages an efficient flexible tree matching solution that we identified. Section ?? describes the locator benchmark we designed and implemented. Section ?? reports on the performance of our approach compared to the state-of-the-art algorithms. Finally, Section ?? presents some perspectives for this work, while Section ?? concludes.

3.3 Background & Related Work

We deliver a novel contribution to the locator repair problem, which has been initially studied in the domain of web testing. In this section, we thus introduce the required background and we describe state-of-the-art approaches to repair broken locator, and in particular the literature published in the domain of web test repair.

3.3.1 Introducing Web Element Locators

To detect regressions in web applications, software engineers often rely on automated web-testing solutions to make sure that end-to-end user scenarios keep exhibiting the same behavior along changes applied to the system under test. Such automated tests usually trigger interactions as sequences of actions applied on selected elements and followed by assertions on the updated state of the web page. For example, *"click on button e_1, and assert that the text block e_2 contains the text* `'Form sent'`*"*. To develop such test scenarios, a software engineer can 1. manually write web test scripts to interact with the application, or 2. use record/replay tools [?, ?, ?] to visually record their scenarios. In both cases, the scenario requires to identify the target elements on the page [e_1, e_2] in a deterministic way, which is usually achieved using XPath, a query language for selecting elements from an XML document. For example, let us consider the following HTML snippet describing a form:

```
<form method="post" action="index.php">
    <input type="text"   name="username"/>
    <input type="submit" value="send"/>
```

```
</form>
```

The following XPath snippets describe 3 different queries, which all result in selecting the submit button: `/form/input[2]`, `/form/input[@value="Send"]`, `input[@type="submit"]`. In the literature, such element queries or identifiers are named *locators* [?].

In practice, automated tests are often subject to breakages [?]. It is important to understand that a test *breakage* is different from a test *failure* [?]: a test failure successfully exposes a regression of the application, while a test is said to be broken whenever it can no longer apply to the application (*e.g.*, the test triggered a click on a button e, but e has been removed from the page). While there can be many causes to test breakage, [?] reports that 74 % of web tests break because one of the included locators fails to locate an element in a web page.

3.3.2 Generating Web Element Locators

The fragility of locators remains the root cause of test breakage, no matter they have been automatically generated (*e.g.*, in the case of record/replay tools), or manually written. To tackle this limitation, several studies have focused on generating more robust locators. This includes ROBULA [?], ROBULA+ [?], which are algorithms that apply successive refining transformations from a raw XPath query until it yields an xPath locator that exclusively returns the desired element. Leotta *et al.* [?, ?] also propose Sideral, a graph-based algorithm to generate xPath locators. Sideral requires to specifically train on each application in order to learn what properties are most likely safe to rely on when building xPath locators. While these methods all use ancestors of a given elements as anchor to generate a locator, [?] uses siblings instead, arguing that they make more reliable anchors.

Another work by Yandrapally *et al.* leveraged contextual clues to generate locators [?]. These clues rely mostly on the content surrounding the element to locate which may be problematic in case the content changes. LED [?] uses a SAT solver to select several elements at once, but is never evaluated on different DOM versions. Finally, some works combine several locator generators with a voting mechanism to locate a single element with more robustness [?, ?, ?]. However, all these approaches, which consider a limited set of locator generators, strongly depend on the accuracy of individual algorithms to agree upon a single and relevant locator.

While automatically generating locators can speed up the definition of test cases, it becomes a keystone for visually-generated test cases based on record/replay tools.

In the end, the reliability of test cases built using such a tool depends mostly on the quality of the locators it automatically generates [?].

3.3.3 Repairing Web Element Locators

While some solutions to the robust locator problem, as presented above, aim to prevent locators from breaking, others focus on repairing broken locators. In this context, the repair tool considers *a)* the descriptor of the locator, *b)* the last version of the page on which the locator was still functional (D), *c)* the new version of the page on which the locator is broken (D').

Property Based In this area, WATER [?] and COLOR [?] provide algorithm to fix broken tests using intrinsic properties of the element to relocate. The process of repairing a test involves several steps: 1. running the test, 2. extracting the causes of failure and, 3. repairing the locator, if broken. The last part is particularly challenging. To relocate a locator from one version to another, WATER and COLOR scan all elements in the new version and return the most similar one to the element in the original version with regards to intrinsic properties (*e.g.*, absolute XPath, classes, tag). Hammoudi *et al.* [?] further studied the locator repair part of WATER and found that repairing tests over finer-grained sequences of change (typically commits) contributes to improving accuracy.

Vision Based Using a completely different approach, VISTA [?] is a recent technique that adopts computer vision to repair locators. VISTA falls within the category of computer vision-aided web tests [?, ?, ?]. However, while using computer vision succeeds in repairing most of the *invisible* changes, such solutions tend to fail when the content, the language, or the visual rendering of the website changes. Furthermore, visual-based solutions fail to locate dynamic elements that only appear through user interactions (*e.g.*, a dropdown menu).

Finally, J.Imtiaz et al [?] developed a test repair solution that integrates to several different capture replay tools. While we focus specifically on the locator repair problem, they used and evaluated a more comprehensive test-repair strategy involving the classification of the test script and detected breakages and the extension of the UML Testing Profile specifications to capture more interaction details.

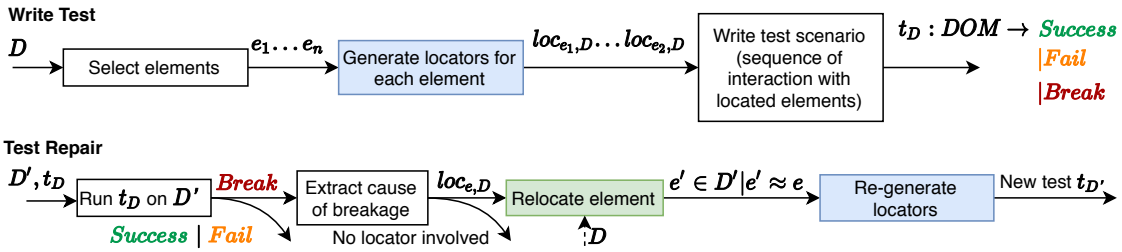


Figure 3.1: Illustration of the locator problem statement in automated tests combining the *robust locator* (in blue) and the *locator repair* (in green) problems.

3.4 Locator Problem Statement

Figure ?? summarizes the steps to follow when writing or repairing a locator in a web test script. When a test breaks, the repairing process generally includes three main steps: 1. extract the cause of the breakage 2. if a locator caused the breakage, the element is first relocated then 3. a new locator is generated/written. Beyond automated tests, this problem can also arise in more general web automation scripts covering web content crawling and *Robotic Process Automation* (RPA), which heavily rely on locators to automate the navigation across web applications.

In this section, we formalize the description of two locator-related problems highlighted in Figure ??, namely the *robust locator* (in blue) and *locator repair* (in green) problems for the general case of web automation scripts.

3.4.1 Problem Notations

We consider that a given web page can change for various reasons, such as 1. content variation, 2. page rendered for different regions/languages, or 3. release of the web application. No matter the cause, we distinguish D and D' as two versions of the same web page observed before and after a change, respectively. More specifically, we define the following similarity notations:

1. $D \approx D'$ if scripts written for D are expected to apply on D';
2. Given 2 web elements $e \in D$ and $e' \in D'$, $e \approx e'$ if e and e' refer to semantically equivalent elements (*e.g.*, the same menu item observed in pages D and D');
3. By extension of (2), given a set of elements $E = e_1...e_n \subset D$ and $E' = e'_1...e'_{n'} \subset D'$, $E \approx E'$ if $n = n'$ and, for each $i \in [1..n]$, $e_i \approx e'_i$.

Based on the above similarity notation, we provide the following definitions:

Definition 3.4.1. *Given a page D, and a set of elements $E = e_1...e_n$, the pair*

*$(loc_{E,D}, eval)$ is a **locator** of E with regard to D if:*

$$eval(loc_{E,D}, D) = E \tag{3.1}$$

*where $loc_{E,D}$ is a descriptor of E and eval an evaluation function that returns a set of web elements from a descriptor and an evaluation context (*e.g.*, a web page).*

In the case of XPath-based locators, the descriptor $loc_{E,D}$ refers to an XPath query describing the elements E in the page D and *eval* the XPath solver.

Definition 3.4.2. *Let mut be a mutation function that transforms the page D into another page D', such as $mut(D) = D'$. mut is said to be a **mutation** of D if $D \approx D'$.*

Definition 3.4.3. *Given a locator $L = (loc_{E,D}, eval)$, L is **robust** to a mutation function mut if:*

$$eval(loc_{E,D}, mut(D)) \approx E \tag{3.2}$$

Finally, we note $\lambda(e)$ the label of the node e in the DOM tree. The label of a node comprises the tag, the attributes and their values and the textual content. However, in the context of ERRATUM, we willingly ignore the content as described in section ??.

3.4.2 Problem Statement

Given the above definitions, we can formalize the locator problem statement along with the two following research questions.

RQ 3.4.1. ***Robust Locator**. For any subset of elements on a given page D, how to automatically generate locators that are robust to mutations of D?*

When evaluating a locator on a new page D', the only available information to describe the targeted element is the descriptor $loc_{E,D}$, which often remains insufficient (cf. state-of-the-art techniques).

On the other hand, in the context of *locator repair*, the original page D from which $loc_{E,D}$ was built is available. Thus, using definition **??**, this piece of information allows to locate the originally selected elements $eval(loc_{E,D}, D) = E$.

RQ 3.4.2. ***Locator Repair**. Given two pages D, D', such that $D \approx D'$ and a set of elements $E \in D$, how to locate the elements $E' \approx E$ in D'?*

To the best of our knowledge, existing solutions to both *robust locator* and *locator repair* focus on the restricted case of $|E| = 1$.

Once the *locator repair problem* is solved (*i.e.*, E' are correctly located), we need to generate new locators, which brings us back to the situation of the robust locator problem (cf. RQ. **??**).

In this article, we thus present a novel approach to solve the locator repair problem.

3.5 Repairing Locators with ERRATUM

The previous section formalized both *robust locator* and *locator repair* problems. The approach we report in this article, ERRATUM, therefore matches the DOM trees of 2 versions of a web page to solve the *locator repair* problem. Several tree matching solutions exist in the literature, such as *Tree Edit Distance* (TED) [**?**] or tree alignment [**?**]. This section therefore motivates and explains how ERRATUM leverages tree matching to repair locators, before discussing the choice of a tree matching implementation fitting ERRATUM's requirements.

3.5.1 Applying Tree Matching to Locator Repair

Embedding tree matching allows ERRATUM to leverage the tree structure in the same way an XPath-based solution would, while offering the flexibility of a more statistic-based solution. Intuitively, a tree matching algorithm should consider all easily identifiable elements on a page (elements with rare tags, unique classes, ids, or other attributes) as *anchors* to relocate less easily identifiable elements.

Figure **??** illustrates the benefits of a more holistic approach using tree matching. In the example, the locator of element `a` (in blue) breaks because the mutations between D and D' entails a change in its absolute XPath (`/body/div/a`). Attempting to repair such a broken locator by relying on the properties of the original element alone (state-of-the-art approaches like [**?**, **?**]) is often challenging and can easily lead to a mismatch. By using tree matching (cf. right-bottom of Figure **??**), matching the parent of the element to locate (`div#menu`) brings instead a strong contextual clue to accurately relocate the element $\texttt{a}_1$' whose locator was broken [**?**].

Formally, given a pair of page versions D and D', we:

1. parse D and D' into DOM trees T and T'. Consequently, $nodes(T)$ is the set of elements in the DOM tree T;

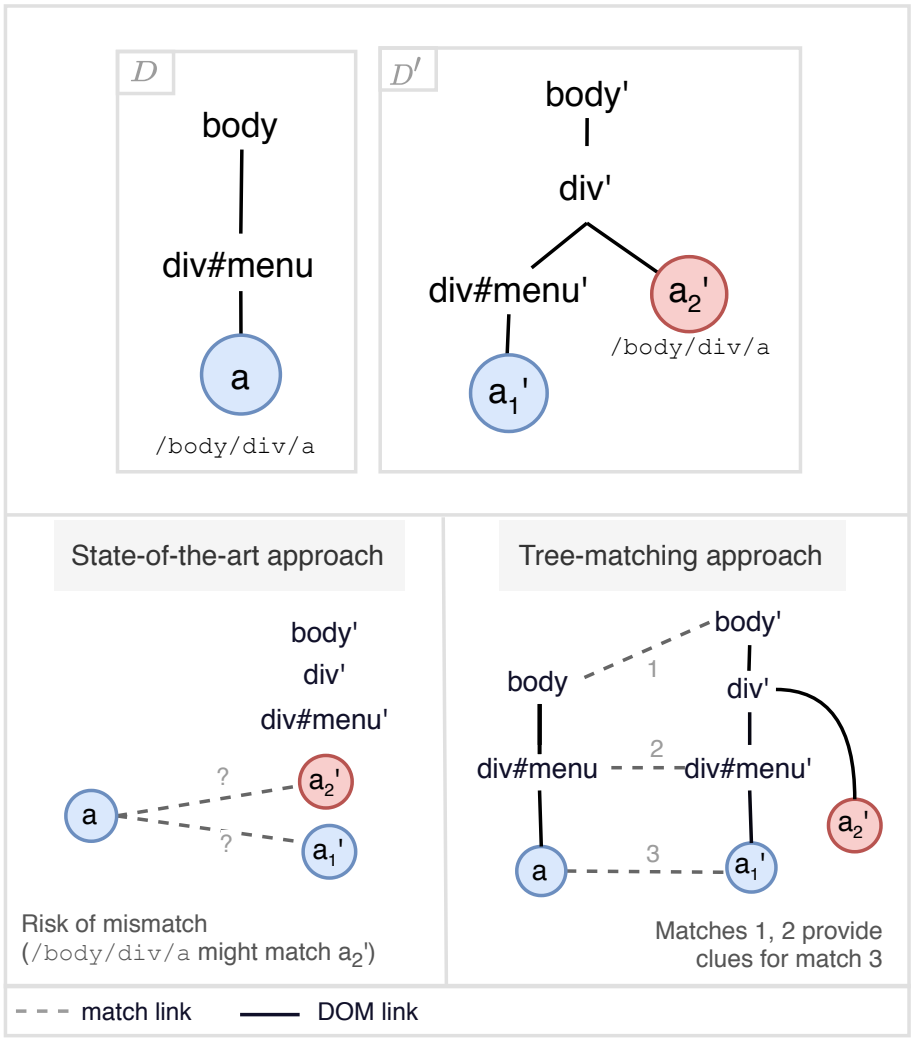


Figure 3.2: State-of-the-art Vs. tree matching locator repair.

2. apply tree matching to T, T' yielding a matching $M \subset nodes(T) \times nodes(T')$. If the resulting matching M is accurate, then $\forall (e, e') \in M, e \approx e'$;
3. use the resulting matching M to repair the broken locator(s).

Regarding the test repair process illustrated in Figure **??**, our approach thus fits in the block "relocate element" (in green) by matching the elements of D in D' and reporting the relocated element. Thus, once the element is relocated using tree matching—*i.e.* ERRATUM found $e' \in D' | e \approx e'$—we only need to generate a new locator $loc_{e',D'}$ to achieve the test repair process. This task can be performed using solutions to the robust locator problem, like ROBULA [**?**], and is therefore considered as out of the scope of this article.

3.5.2 Integrating a Scalable Tree Matching Algorithm

The state-of-the-art approach to match two trees is *Tree Edit Distance* (TED) [**?**]. When comparing two trees T and T', TED-based approaches rely on finding the optimal sequence of relabels, insertions and deletions that transforms T into T'. Unfortunately, TED might be unsuitable to match real-life web pages due to two core restrictions [**?**]: 1. if two nodes e and e' are matched, the descendants of e can only match with the descendants of e', and 2. the order of siblings must be preserved. Furthermore, TED is computationally expensive ($O(n^3)$ for the worst-case complexity [**?**]) and, more practically, our preliminary experimentation has shown that applying the state-of-the-art implementation of TED, named APTED [**?**], on the *YouTube* page takes several minutes. We believe that, in addition to qualitative restrictions, such computation times are not acceptable when periodically repairing locators on real websites.

Further studies of TED proposed to improve computation times [**?**, **?**, **?**], but at the cost of even more restrictive constraints on the produced matching (*e.g.*, the tree alignment problem [**?**] restricts the problem to transformations where insertions are performed before deletions).

To the best of our knowledge, the only contribution that provides a solution to the general (restriction-free) tree-matching problem is the *Flexible Tree Matching* (FTM) algorithm [**?**]. FTM models tree matching as an optimization problem: given two trees T and T' how to build a set of pairs $(e, e') \in T \times T'$ such that the similarity between all selected node pairs is maximal. The similarity used by FTM combines both the labels and the topology of the tree.

However, as shown in [**?**], the theoretical complexity of FTM is high ($O(n^4)$)

and the implementation of FTM was shown to take more than an hour to match a web page made of only 58 nodes, while the average number of nodes on a web page observed in our dataset is 1,507. Consequently, we believe that such computation times make FTM unpractical in the context of locator repair.

3.5.3 Matching DOM Trees by Similarity

Given the limitation of FTM, ERRATUM integrates a *Similarity-based Flexible Tree Matching* (SFTM) algorithm, which is an extension of state-of-the-art FTM to improve the computation times of FTM without any restriction on the resulting matching [**?**].

In the context of ERRATUM, the SFTM algorithm assumes that, given a web page, several elements are easily identifiable by considering their intrinsic properties. The algorithm first assigns scores to all possible matches between nodes from the two trees based on their label and only then uses the topology of the trees to adjust these scores.

Almost all existing tree matching algorithms rely first and foremost on the topology of the trees. Conversely, SFTM relies mostly on the labels of the trees and only makes use of topology in a second step, to fine-tune the already computed scores. Intuitively, matching two sets of labels is significantly easier than trying to match trees, which is the reason why SFTM achieves such competitive performances. The only trade-off of this approach is that it requires the labels of the trees to be highly differentiating (*i.e.*, carry a lot of information). Fortunately, this is the case for the great majority of web pages.

We walk through the key steps of the SFTM algorithm we integrated in ERRATUM by using HTML snippets reported in Figure **??**. The figure provides two versions (D and D') of a simplified HTML code sample extracted from the homepage of the famous search engine *duckduckgo.com*. In this example, our purpose is to relocate $\mathrm{a}_1 \in D$ with $\mathrm{a}'_1 \in D'$.

Unlike the state-of-the-art matching algorithms, SFTM first tries to match elements in D' whose labels are similar to D, before using these matched elements to adjust the similarity of surrounding elements in the tree. For example, the *similarity scores* of the tuple (a_1, a'_1) links will increase as their direct parents (div_3, div'_3) are matched with confidence. Figure **??** summarizes the SFTM algorithm's key steps and the remainder of this section provides an overview of its integration in ERRATUM (cf. green box in Figure **??**). The interested reader can refer to our technical report [**?**]

(a) Original document D.

```
<div class="content-info__item"> (div1)
    <div class="item__title">...</div> (div2)
    <div class="item__subtitle"> (div3)
        ...
        <a href="/plugins">Plugins</a> (a1)
    </div>
</div>
```

(b) Updated document D' from D.

```
<div class="items-wrap"> (div'4)
    <div class="item"> (div'1)
    <div class="item__title">...</div> (div'2)
        <div class="item__subtitle"> (div'3)
            ...
            <a href="/extensions">Extensions</a> (a'1)
        </div>
    </div>
    <div> (div'5)
        ...
        <a href="/newsletter">Newsletter</a> (a'2)
    </div>
</div>
```

Figure 3.3: Two versions of an HTML snippet extracted from the homepage of *duckduckgo.com*.

for an exhaustive description of our SFTM algorithm, whose applications go beyond the context of repairing broken locators.

Step 1: Node Similarity Elements of DOMs D and D' are compared. The first step consists in computing an *initial similarity* $s_0 : D \times D' \to [0, 1]$. For each pair of nodes $(e, e') \in D \times D'$, $s_0(e, e')$ measures how similar the labels of e and e' are. In HTML pages, the label of a node $e \in D$ that we use is the set of tokens obtained from applying a tokenizer to the HTML code describing e. This may include the type of the HTML element, its attributes, and eventually the raw content associated to this element—*i.e.*, thus ignoring the content from the child elements.

Example 3.5.1. *The* label *computed for* div_1 *(cf. Figure* **??***) includes the following tokens:* $\{$`div`, `class`, `content-info__item`$\}$.

To compute s_0, SFTM first indexes the labels of each node of D. The idea of this step is to prune the space of possible matches by pre-matching nodes with similar labels. When indexing, to improve the accuracy of s_0, we apply the *Term Frequency-Inverse Document Frequency* (TF-IDF) [**?**] formula to take into account how rare each token is.

Example 3.5.2. *Following our previous Example* **??***, when considering the match* (div_1, div'_1)*:*

1. *token* `div` *will yield very few similarity points, since it is included in the labels of almost all the nodes,*
2. *token* `content-info__item` *will increase the score significantly, as it only appears once in both documents.*

In general, very common tokens bring very few information to the relevance of a given match, while they cause a significant increase of potential matches to consider. That is why, in order to reduce the computation times, the algorithm rules out most common tokens.

Step 2: Similarity Propagation The initial similarity s_0 only takes into account the labels of nodes. In this second step, the idea is to enrich the information contained in s_0 by leveraging the topology of the trees D and D'.

Example 3.5.3. *In Example* **??***, it is hard to choose the correct match* $m_1 = (a_1, a'_1)$ *over the incorrect one* $m_2 = (a_1, a'_2)$ *by only considering labels, since all three elements share the same set of tokens:* $\{$`a`, `href`$\}$*. In the similarity propagation step, we leverage*

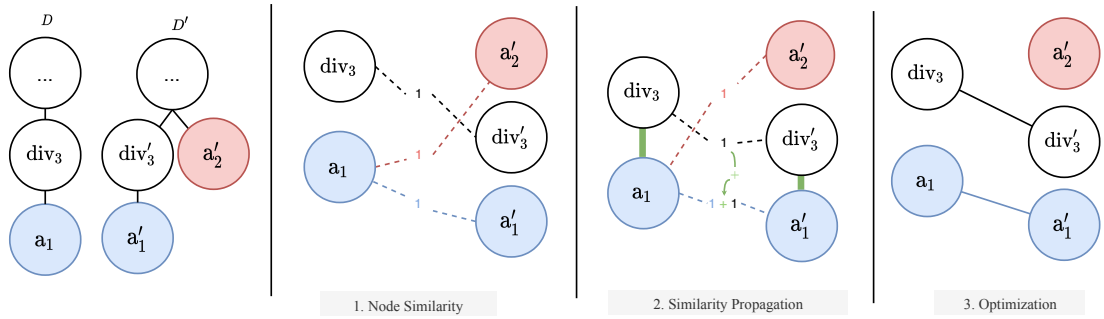


Figure 3.4: Key steps followed by our *Similarity-based Flexible Tree Matching* (SFTM) algorithm.

the fact that the parents of a_1 *and* a'_1 *are similar to increase the similarity between* a_1 *and* a'_1*, thus preferring* m_1 *over* m_2*.*

In general, for each considered match $(e, e') \in D \times D'$, the parents of e and e' gets more similarity points if e and e' are similar and inversely, $s(e, e')$ is increased if the parents of e and e' are similar (with regard to s_0). We call s the final similarity produced by this step.

Step 3: Optimization Producing the optimal matching with regards to the computed similarity means selecting the full set of matches such that each element of D is matched with at most one element of D' and the sum of similarity scores of the selected matches is maximum.

In order to approximate the optimal set of matches, SFTM implements the Metropolis algorithm [**?**]. The idea is to randomly walk through several possible configurations (set of matches) to converge towards the optimal one.

At the end of the optimization step, the SFTM algorithm yields a full matching $M \subset D \times D'$ comprising matches between nodes of D and D'. These matches can be analyzed by ERRATUM to locate broken locators and fix them by generating new locators in the target document D'.

Best and worst case scenarios The greatest strength of ERRATUM's approach is to handle variations of labels—*e.g.*, renaming a class, adding an id, removing an attribute. This is because of the fuzzy nature of the first similarity steps of SFTM: as long as two elements' label share enough rare tokens, they will match. For elements that have few information in their label (*e.g.*, a bare *div* tag with no attributes) the algorithm will still be able to rely on the parents and children of the node whose label may contain more discriminative information.

The worst case scenario happens when making structural changes around nodes with with labels containing few information. For example, if the content of a bare *div* node is moved to another *div* node, matching the first *div* node accurately will become very challenging. More than that, since the node that was moved contains high amount of information, the matching of this node will propagate to its new parents, thus increasing the chances of mismatch in the surrounding of the moved node.

3.6 The Robust Locator Benchmark

In the context of this article, we are interested in covering the following research questions:

RQ 3.6.1. *How does* ERRATUM *perform in solving the locator repair problem (cf. RQ.* ??*) when compared to state-of-the-art solutions?*

RQ 3.6.2. *What are the factors influencing the accuracy of WATER and* ERRATUM*?*

RQ 3.6.3. *How quickly can* ERRATUM *repair broken locators when compared to state-of-the-art solutions?*

This section, therefore, describes the benchmark we propose to assess these questions.

3.6.1 Evaluated Locator Repair Solutions

We compare two solutions: 1. ERRATUM, our approach to repair broken locators by leveraging flexible tree matching, and 2. WATER, the reference implementation of a locator repair technique applied to web test scripts [**?**].

The original algorithm of WATER analyses a given test case, finds the origin of the test breakage, and suggests potential repairs to the developer. In the context of this article, we are interested in the most challenging part of the algorithm: the part that repairs broken locators, if needed. Given the originally located element $e \in D$, WATER attempts to find $e' \in D'$ such that $e' \approx e$ by scanning over all elements in D' such that $tag(e') = tag(e)$ and selecting the elements most similar to e. The similarity between two elements e_1, e_2 used by WATER mostly consists in computing the Levenshtein distance between the absolute XPaths of both elements

($Levenshtein(XPath(e_1), XPath(e_2))$) combined with other element properties similarity (*e.g.*, visibility, z-index, coordinates). In our evaluation, we re-implemented this part of the WATER algorithm to compare its performance to ERRATUM.

We initially considered VISTA [?] as a baseline, even though the approach they use (computer vision) is radically different from ERRATUM and WATER. However, despite our efforts, we failed to run their implementation and received no answer when trying to contact the authors.

Note that, in this evaluation, we focus on *single-element locator* cases of the locator repair problem (we only try to repair *single-element locators*). The reasons for this decision are: 1. The state-of-the-art solutions to both repair and robust locator problems only treat this case and in particular, WATER can only repair locators locating a single element, 2. ERRATUM reasons on the whole trees, so locating several independent elements is done the same way as locating a group of elements.

3.6.2 Versioned Web Pages Datasets

In the remainder of this article, we propose two datasets to compare ERRATUM and WATER against potential evolutions of web pages. Given two versions of the same page D, D', and a set of elements $E \subset D$, the locator repair problem consists in locating a set of elements $E' \subset D'$, such that $E' \approx E$. To evaluate the performance of a locator repair tool, we thus need what we call a **DOM versions dataset**: a dataset of pairs (D, D'), such that $D \approx D'$.

A DOM version dataset is also required to evaluate solutions to the robust locator problem. To build such a dataset, previous works on locator repair [?, ?] and robust locator [?, ?, ?] manually analyzed different versions of a few open source applications (like Claroline, AddressBook or Joomla). These evaluations are significantly limited in size (never beyond a dozen of websites considered) and hard to reproduce since the exact versions of the open source applications used are often not provided or available.

In our study, we therefore introduce the first large-scale, reproducible, real-life *DOM versions dataset* that can be used to assess locator repair solutions, and is composed of two parts:

1. A **MUTATION dataset** [?] generated by applying random mutations to a given set of web pages (see Section ??),
2. A **WAYBACK dataset** collects past versions of popular websites from the Wayback API (see Section ??).

Then, for each pair (D, D') in the dataset, our experiments consist of selecting a

set of elements to locate in D and in comparing both ERRATUM and WATER trying to find the corresponding element on D'.

Table **??** describes both datasets in terms of:

1. # **Unique URLs**: the number of unique URLs among the total of version pairs in the dataset. The duplication is due to the fact that there can be several mutations or successive versions of the same web page. In the case of the WAYBACK dataset, more popular websites are more represented (see Section **??**);
2. # **Version pairs**: the number of considered pairs of web pages (D, D'),
3. # **Located elements**: the number of elements $e \in D$ that any solution should locate in D'.

Table 3.1: Description of the MUTATION & WAYBACK datasets.

Dataset	MUTATION	WAYBACK
# Unique URLs	650	64
# Version pairs	3,291	2,314
# Located elements	49,305	34,421

The two datasets we provide are complementary. Since the MUTATION dataset is generated by mutating elements from an original DOM D, the ground truth matching between D and its associated mutation D' is known to easily evaluate the solution on a very large amount of version pairs. However, since the versions are artificially generated, this dataset is synthetic and, as such, might not entirely reflect the actual distribution of mutations happening along a real-life website lifecycle.

Then, the WAYBACK dataset is composed of real website versions mined from the Wayback API: an open archive that crawls the web and saves snapshots of as many websites as possible at a rate depending on the popularity of the website.[3] In the WAYBACK dataset, mutations between D and D' are not synthetic, but as a result, the ground truth matching between D and D' is unknown. In our evaluation, we thus had to manually label a sample of the results obtained on this dataset, which limits the scalability of the experiment compared to the MUTATION dataset.

The following sections provide more details on how both datasets were built.

Building the MUTATION dataset.

We extend the technique we introduced to generate a MUTATION dataset in [**?**]. The mutation dataset is built by applying a random amount of random mutations to a set

[3] `https://archive.org/help/wayback_api.php`

Table 3.2: Mutations applied in the MUTATION dataset [?].

Type	Mutation operators
Structure	`remove, duplicate, wrap, unwrap, swap`
Attribute	`remove, remove words`
Content	`replace with random text, change letters, remove, remove words`

of original web pages: for each original DOM D, 10 mutants are created by applying mutations to D. Since the mutations applied to D to construct each mutant D' are known, the ground truth matching between D and D' is also known. Knowing the ground truth matching on the mutation dataset allows us to evaluate our locator repair solution on a very large dataset. Table **??** describes the set of DOM mutations that can be observed along evolution of a web page.

The original websites from which mutants were generated were randomly selected from the Top 1K Alexa. Figure **??** depicts the distribution of DOM sizes in this synthetic dataset.

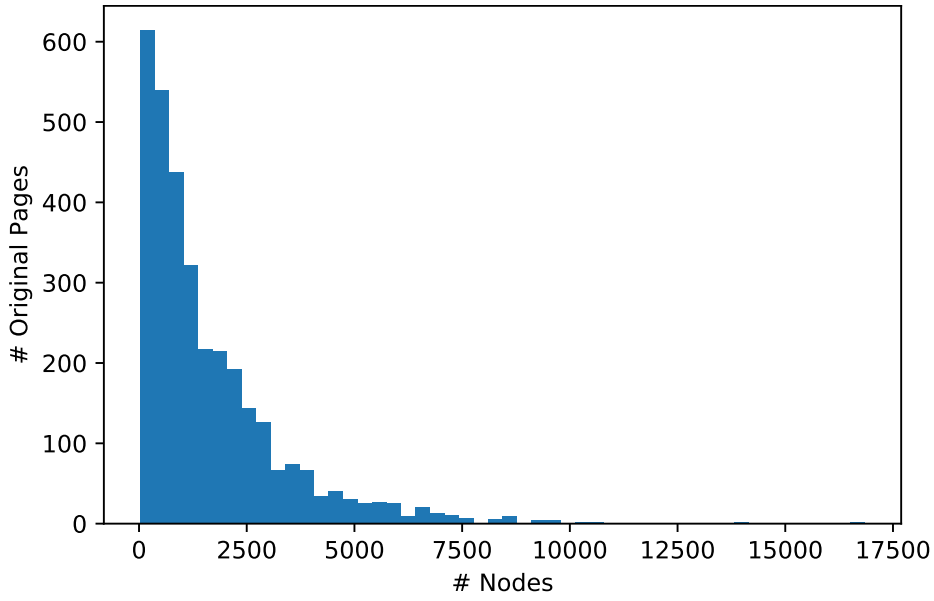


Figure 3.5: Distribution of DOM sizes (in number of nodes) in the MUTATION dataset.

This dataset was built with an automation tool that we made available along with its source code **??**. From a given list or source URLs, our tool creates a dataset of randomly mutated web pages following the above-described approach.

Buidling the WAYBACK dataset.

This dataset encloses a list of (D, D') DOM pairs where D and D' are two versions of the same page (*e.g.*, *google.com* between 01/01/2013 and 01/02/2013). Two versions can be separated by different gaps in time. In this section, we explain how we used the Wayback API to build this dataset. The Wayback API can be used to explore past versions of websites. The two endpoints we used to build the dataset can be modeled as the following functions:

$$\begin{aligned} versionsExplorer :: & \quad (url, duration) \rightarrow & timestamp[] \\ versionResolver :: & \quad (url, timestamp) \rightarrow & document \end{aligned}$$

The *versionsExplorer* retrieves the list of available snapshots between two dates, while the *versionResolver* returns the snapshot of a given *url* at the requested timestamp.

Using these endpoints, for each website URL considered, we:

1. retrieved the timestamps of all versions between 2010 and today using the *VersionExplorer*,
2. generated a list of all pairs of timestamps with one of following differences in days ($\pm 10\%$): [7, 15, 30, 60, 120, 240, 360],
3. picked up to $1,000$ random elements from the list of timestamps pairs,
4. resolved each selected timestamp pair using the *versionResolver*.

Similarly to the MUTATION dataset, the URLs we fed to our algorithm were taken from the Top 1K Alexa. Since both datasets are based on the same set of URLs (taken from Alexa), the distribution of the WAYBACK dataset is very similar to the MUTATION one (cf. Figure **??**).

Selecting the elements to repair.

ERRATUM and WATER operate in different ways. ERRATUM takes two trees (D, D') and returns a matching between each element of the trees, thus solving any possible broken locator between D and D'. The algorithm extracted from WATER is a more straightforward solution to the locator repair problem as formally described (cf. Section **??**): it takes a pair of DOM versions (D, D') and an element $e \in D$ as input and returns an element $e' \in D'$ (or *null* if it fails to find any candidate for the matching).

Consequently, in the case of the WATER algorithm, the following question arises: given a pair (D, D') taken from the DOM version dataset, which elements of D should be picked for repair? Ideally, we would try to locate every element of D in D' to

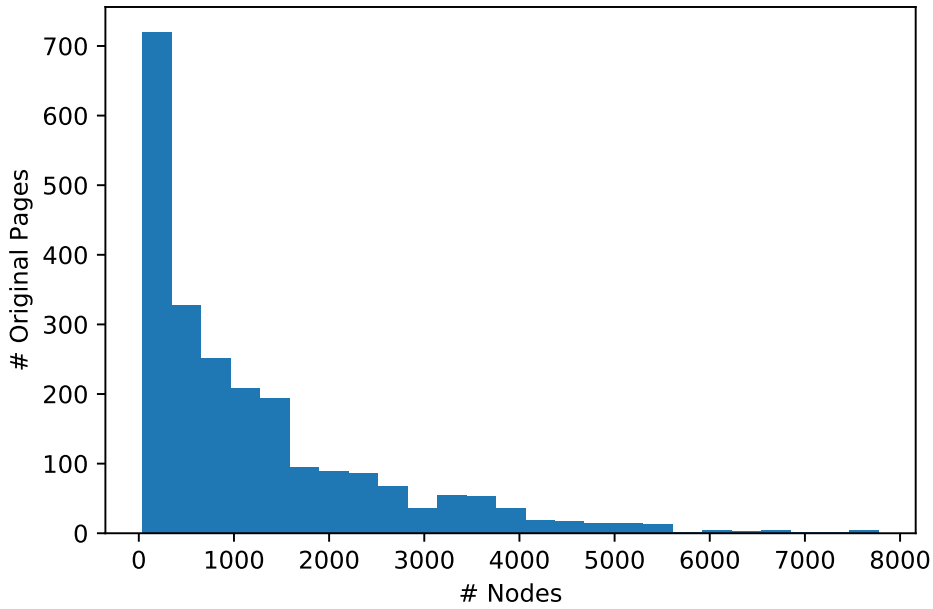


Figure 3.6: Distribution of DOM sizes (in number of nodes) in the WAYBACK dataset.

obtain a comprehensive comparison with ERRATUM. Unfortunately, the computation times of WATER make it impractical to locate every single element from D in D'. Selecting realistic targets for locators is a non-obvious task since many elements in the DOM would not be targeted in a test script (*e.g.*, large container blocks, invisible elements, aesthetic elements). Therefore, for each version pair, we randomly select up to 15 clickable elements from D. We focus on clickable elements as this is the most common use case for web UI testing (to trigger interactions), and WATER has specific heuristics to enhance its accuracy on links. By considering clickable elements, we 1. make sure to choose realistic elements, and 2. compare to WATER on its most typical use case.

Regarding the sample size, considering 15 elements per web page leads to selecting 34,000+ elements in both datasets. As the average number of nodes per web page in each dataset is around $1,500$, this means that there are more than 3.6M candidate locators for repair in each dataset. Therefore, the confidence interval at 95 % of the measurements applied to the 34K sample of located elements is 0.5 %.

3.6.3 Evaluating of the Matched Elements

On the MUTATION dataset, the signatures attributes are preserved after mutations (but ignored when applying either locator repair solution), thus providing the ground truth matching between the DOMs of a version pair.

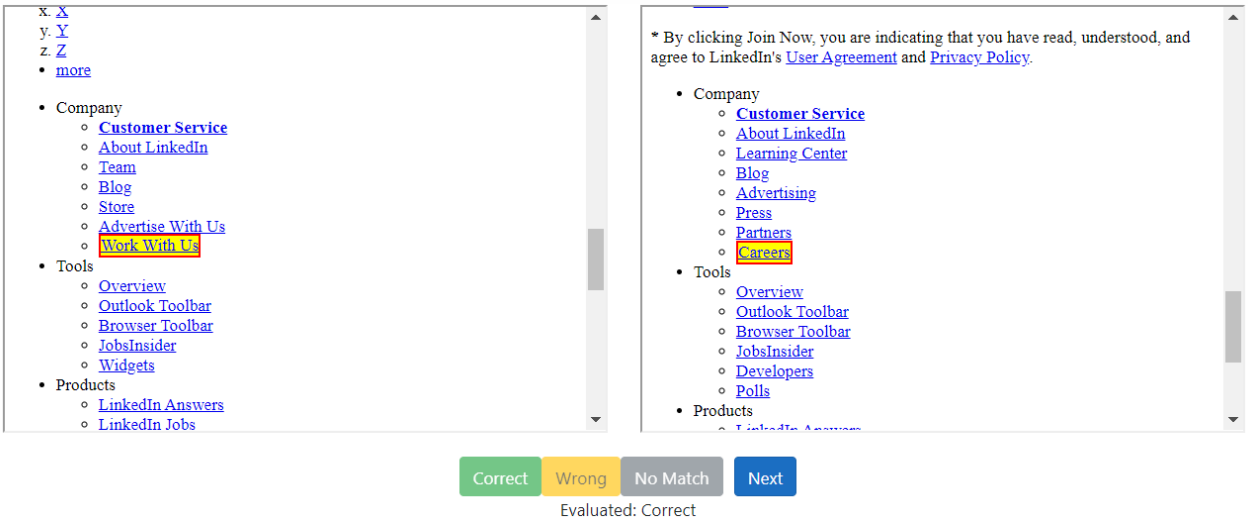


Figure 3.7: Labeling a given element matched by ERRATUM on two versions of the Linkedin homepage. The screenshot comes from the visual matching application we created to manually label disagreements between ERRATUM & WATER.

For the WAYBACK dataset though, this information is not available. For each version pair (D, D'), the evaluation of both solutions yields to a list of suggested matching $(e, e'_{ERRATUM})$ and (e, e'_{WATER}) where $e \in D$ and $e'_{ERRATUM}, e'_{WATER} \in D'$. In both cases, e' may be null in case no matching was found. Given the above situation, the labeling process consists of determining whether the matching element of e is $e'_{ERRATUM}$, e'_{WATER}, or neither. In many cases, $e'_{ERRATUM} = e'_{WATER}$ (consensus). We choose to focus our manual labeling effort on cases where WATER and ERRATUM disagree and assume that both solutions are right otherwise.

Thus, to label the disagreements between ERRATUM and WATER, we developed a web application (cf. Figure ??) to display the identified elements on both versions of the DOM version pair and label the matching as either correct or wrong.

When we defined the similarity equivalence between two elements (cf. Definition ??), we mentioned the potential subjective part of the measure. To lessen this subjective part and label the proposed matchings as objectively as possible, we systematically recommended the following guidelines:

1. Sometimes, matched elements are not visible (it happens when the visibility of some parts of the page is triggered dynamically). In this case, if elements in both versions are not visible, the locator is skipped, otherwise, the matching is considered as mismatch;
2. Sometimes, a link appears in different locations on the website (often sign-in

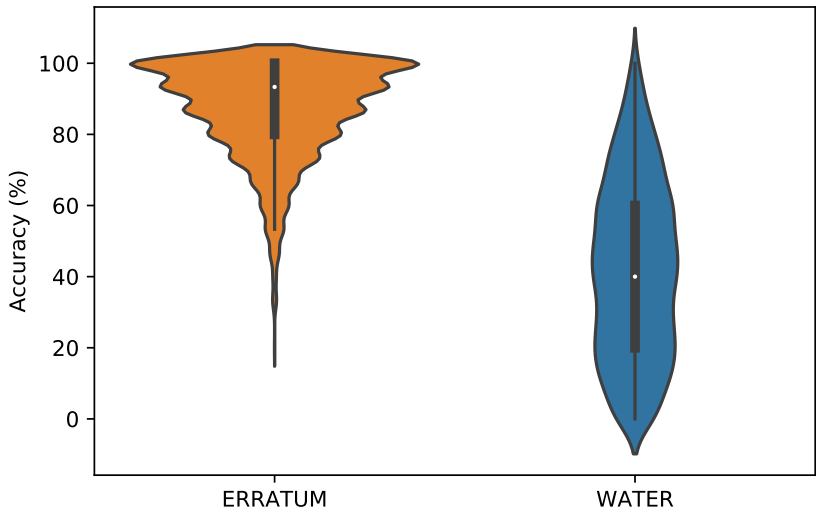


Figure 3.8: Accuracy distribution of ERRATUM and WATER on the MUTATION dataset.

links). Matching two such links from different locations is considered as wrong even though the two links might be assumed to have a similar semantic value. Therefore, we always consider the surrounding of the located element to judge whether the matching is correct or mismatch.

3.7 Empirical Evaluation

This section evaluates locator repair solutions along with two criteria, accuracy and performance, to answer our research questions.

3.7.1 Evaluation of Repair Accuracy

In this section we answer RQ.??: How does ERRATUM perform in solving the locator repair problem (RQ. ??) when compared to state-of-the-art solutions?

Repair accuracy on the MUTATION dataset. Figure ?? summarizes the distribution of the accuracy of ERRATUM and WATER as a violin plot over the 3,291 version pairs of our MUTATION dataset. For each version pair (D, D'), the reported accuracy ratio corresponds to the ratio of the 15 selected elements from D that are accurately located in D'. The figure shows

There are two ways a repair solution can fail to locate an element $e \in D$ in D': 1. a mismatch, when the original element $e \in D$ has been matched to the wrong element $e' \in D'$, or 2. a no-match, when the algorithm does not manage to locate e

in D'. In case of failure, a no-match is always preferred to a mismatch, since a no-match alerts the developer about failure. Thus, considering the two classes of errors on the MUTATION dataset, Table ?? summarizes the ratio of no-match and mismatch reported by both solutions. In particular, the data shows a significant advantage in favor of ERRATUM when it comes to reducing locator mismatches, compared to WATER.

Table 3.3: Errors distribution of ERRATUM and WATER on the MUTATION dataset.

	ERRATUM		**WATER**	
correct	42,876	(87.0%)	20,740	(42.1%)
mismatch	4,420	(9.0%)	26,820	(54.4%)
no-match	2,009	(4.0%)	1,745	(3.5%)
Total:	49,305	(100%)	49,305	(100%)

To further understand which factors influence the accuracy of ERRATUM and WATER (RQ.??), we studied the evolution of accuracy according to three factors: 1. the type of mutations applied, 2. the size of the DOM (number of nodes) of the original page D, and 3. the mutation ratio applied to the original page D to obtain the mutant D'.

First, to assess the impact of the mutation type on the accuracy of ERRATUM and WATER, we used a constrained version of the MUTATION dataset with only one mutation operation applied for each mutant. In the original MUTATION dataset, a mutant D' of a page D is obtained by picking a random number l of random nodes $n_1, n_2...n_l \in D$ and applying a random mutation type (cf. Table ??) to each node. In the constrained version, we use the same original pages D, but select a single random mutation operation per mutant D'. We then apply the mutation operation to l randomly selected nodes: $n_1, n_2...n_l$. For each original page D, the result is a list of mutants such that each mutant D' was obtained using only one mutation operation on a random amount of random nodes. Figure ?? depicts the sensitivity of both locator repair solutions on this alternative dataset. The vertical lines on top of each bar represent the confidence interval. The figure highlights that ERRATUM is almost exclusively sensitive to structural mutations. In particular, the average accuracy of ERRATUM is not sensitive to content mutations on the page, which is expected since the algorithm ignores the content of the nodes by default. The very low sensitivity of ERRATUM to attributes related mutations is more surprising as attributes account for a major part of the similarity metric of the algorithm. For this reason, we believe that

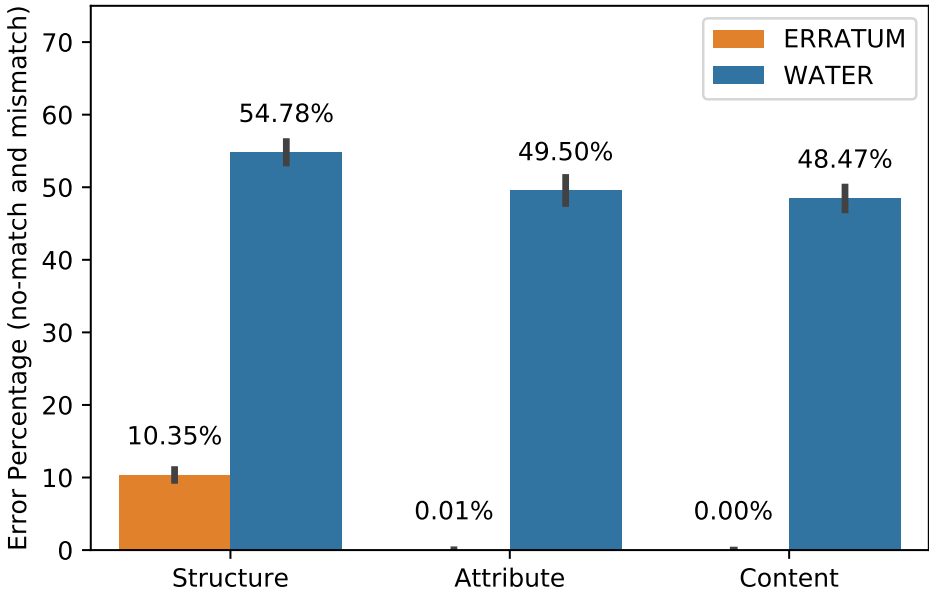


Figure 3.9: Error percentage according to the mutation type.

the mutation of attributes might have more impact when combined with structural mutations, which does not happen in the constrained MUTATION dataset.

Then, regarding the impact of the size of the DOM, our analysis concludes that WATER loses accuracy when the number of nodes increases (cf. Figure **??**), while ERRATUM exhibits a more stable performance. The Spearman correlation coefficient between the error ratio of WATER and the size of the DOM is $\rho = 0.41$ compared to 0.28 for ERRATUM. Interestingly, while ERRATUM correlates rather strongly ($\rho = 0.46$) with the percentage of mutation between the two DOM versions, WATER shows almost no correlation with the same variable ($\rho = 0.12$). It means WATER is surprisingly not impacted by the amount mutations between the versions. The dependency to the mutation ratio of ERRATUM is easily explainable: for each match, ERRATUM indirectly relies on structural and textual similarities in the whole DOM which means mutations anywhere in the DOM could theoretically impact the scores on which ERRATUM relies to compute a matching. Conversely, WATER approach is fundamentally more local to the element to match. We believe these are key insights in understanding the limitation of WATER when compared to ERRATUM. For each element $e \in D$ to locate, WATER searches through all same-tag elements in D' (the *candidates*) and picks the closest one to D, with respect to WATER's chosen similarity metric. We believe that the sensitivity of WATER to the number of nodes comes from the fact that the number of *candidate* matchings for a given element e tends to grow with the size of the DOM, which increases the complexity of

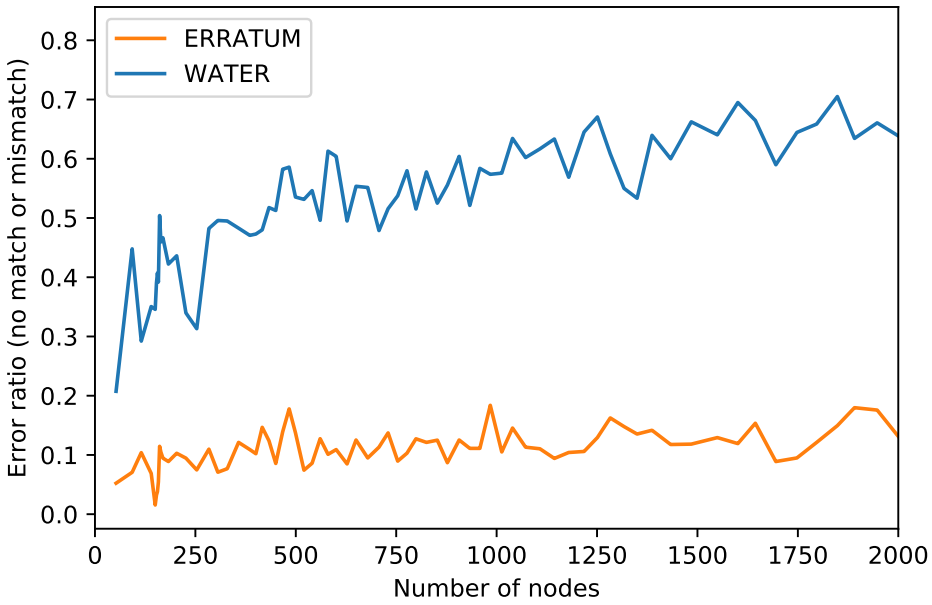


Figure 3.10: Errors rate evolution according to DOM size.

the ordering-by-similarity task. Conversely, additional nodes provide more "anchor" points to ERRATUM, partially compensating the increase in possible combinations.

Finally, regarding the impact of the mutation ratio ($\frac{\#mutations}{\#nodes}$), Figure **??** reports on how ERRATUM and WATER's errors evolve when increasing the number of mutations ($\#mutation$) on the original page D. The figure contains more information than most common box plots, in particular: the stars indicate the average ratio, the horizontal orange lines, the medians whose values also appear above the boxes. As expected, both solutions lose accuracy when the mutation ratio increases, but one can still observe that ERRATUM demonstrates a significant advantage over WATER, no matter the mutation ratio, and exhibiting only 20% of errors on average (against 67% for WATER) when the ratio of mutation exceeds 20% of the nodes.

Repair accuracy on the WAYBACK dataset. Since the WAYBACK dataset does not provide any ground truth matching, we had to manually label the results of the evaluation. We ran both algorithms on the same 34,421 elements. For each element $e \in D$, ERRATUM and WATER returned e'_S and $e'_W \in D' \cup \emptyset$, respectively. In 49.0% of cases, ERRATUM and WATER agreed on a matching element ($e'_S = e'_W \neq \emptyset$). In 13.6% of cases, no solution found a matching element ($e'_S = e'_W = \emptyset$). In 37.4% of cases, ERRATUM and WATER disagreed on the matching element ($e'_S \neq e'_W$ and $(e'_S, e'_W) \neq (\emptyset, \emptyset)$).

A sample of 366 matchings out of the 14,784 disagreements where labelled by web testing experts, which corresponds to a 5% confidence interval at 95%. Table **??**

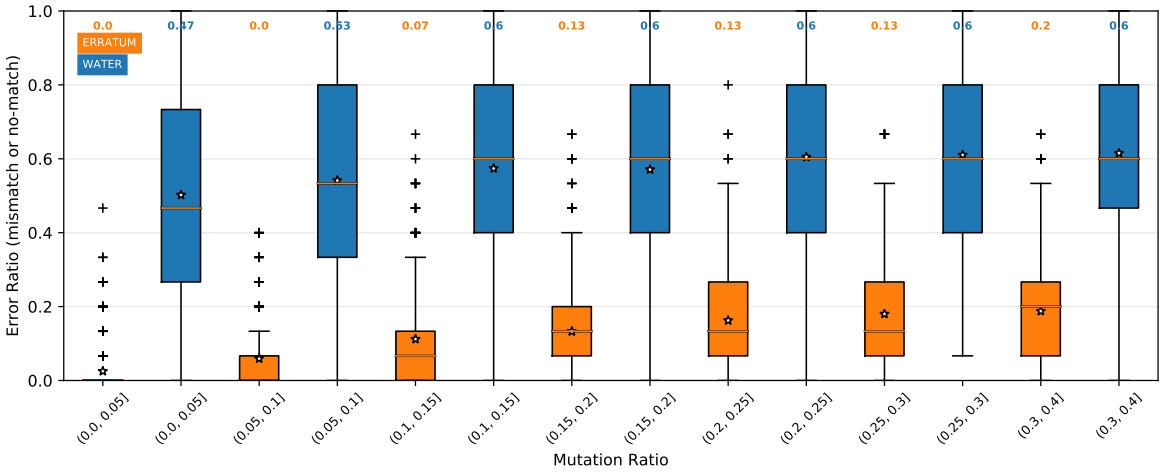


Figure 3.11: Errors rate evolution according to the mutation ratio.

reports on the results of the manual labeling (for disagreements), thus assuming that both WATER and ERRATUM are correct whenever they agree.

Table 3.4: Confusion matrix on the WAYBACK dataset.

		ERRATUM			
		correct	mismatch	no-match	
WATER	correct	49.0%	1.5%	1.4%	51.9%
	mismatch	26.5%	5.5%	3.3%	35.3%
	no-match	2.8%	1.9%	8.1%	12.8%
		78.3%	8.9%	12.8%	

We further investigated the causes of no-match cases reported by ERRATUM to assess if these specific cases could be matched by experts. As part of the WAYBACK experiment, we thus included ERRATUM's no-match cases in our labelling application (cf. Figure ??) and requested the participants to eventually propose a matching element if a no-match was reported by ERRATUM. The result of this evaluation, summarized in Figure ??, highlights that a majority of no-match are accurately labeled as such by ERRATUM, since the participants could not propose a matching element in the target web page. For the few cases where the participants proposed a matching element, we observed that the structure of the DOM tree where subject to many mutations, thus misleading ERRATUM as already observed in Figure ??.

Comparison of repair accuracy. Interestingly, as shown in Table ??, the accuracy of ERRATUM on the WAYBACK dataset ($78.3\,\% \pm 5\,\%$) is 8.7% inferior to the accuracy obtained on the MUTATION dataset (87.0%), while the accuracy of the WATER algorithm is better on the WAYBACK dataset ($51.9\,\% \pm 5\,\%$) than on the

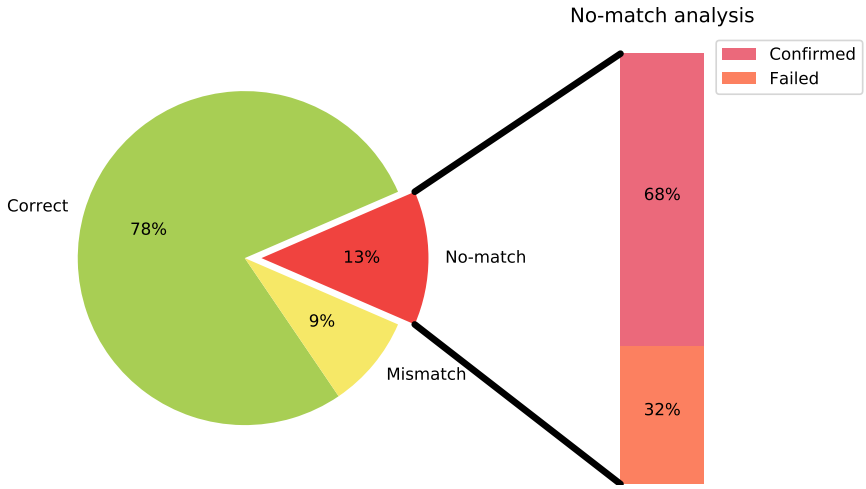


Figure 3.12: Analysis of matches labeled as no-match by ERRATUM.

MUTATION dataset (42.1 %) by 9.8 %. We believe the difference observed between the two datasets is because real-life mutations might not be uniformly distributed. In particular, regarding our sensitivity analysis with regards to types of tree mutations (cf. Figure **??**), one can guess that real-life websites are more subject to *content* and *attribute*-related mutations than *structure*-based mutations (cf. Table **??**), as the former do not affect the accuracy of ERRATUM. However, since we miss the ground truth for the WAYBACK dataset, we cannot assess this hypothesis and the distribution of real-life mutations.

Table 3.5: Accuracy summary across datasets.

	MUTATION	WAYBACK
ERRATUM	87.0%	78.3 ± 5%
WATER	42.1%	51.9 ± 5%

3.7.2 Mutations in the WAYBACK Dataset

To assess the accuracy of ERRATUM on both datasets, we study the nature of the changes occurring between two versions of a given page in the WAYBACK dataset. The changes applied along versions of pages available in the WAYBACK dataset are not labeled, thus lacking a ground truth. The robust locator benchmark (cf. Section **??**) assumes the input datasets as the ground truth to evaluate ERRATUM on the locator repair problem. Conversely, this section assumes the matching algorithm exploited by ERRATUM—*i.e.*, SFTM—to be correct and uses it to label the mutations observed in the WAYBACK dataset. To do so, we estimate the number and types of mutations

between each pair of web pages (D, D') by leveraging the resulting matching $M = sftm(D, D')$. The following table lists the considered mutation types, considering $\forall e \in D$ and $\forall e' \in D'$, where $p(e)$ is the parent of e and $\lambda(e)$ is the label of e:

Label	Mutation	Category
addition	$\nexists e \mid (e, e') \in M$	Structural
removal	$\nexists e' \mid (e, e') \in M$	Structural
move	$(e, e') \in M$ and $(p(e), p(e')) \notin M$	Structural
relabel	$(e, e') \in M$ and $\lambda(e) \neq \lambda(e')$	Relabel

Dataset Consolidation. The robust locator benchmark considers a subset of the WAYBACK dataset, manually labeled by experts. However, during this process, potential inconsistencies observed in rendered web pages were ignored by experts. Therefore, before analyzing the full WAYBACK dataset, one should discard such inconsistencies between unrelated web pages, including cases where: *(a)* one of the versions can be an error page or a redirection page, *(b)* the website may shown "in construction" or may have closed between the two snapshots. For this reason, we apply two heuristics to prepare the dataset by removing all the pairs where:

1. one the two versions has less than 30 nodes, reflecting one of the above inconsistencies. For example, even a minimalist web page, like Google, contains more than 250 nodes;
2. the absolute ratio of sizes between the two versions exceeds 40% which selects approximately 90% of the dataset (see Figure **??**). When this ratio is large, comparing the two pages is likely to be conceptually irrelevant.

While the initial WAYBACK dataset contains 19,161 pairs of web pages, the application of the above rules leads to a consolidated dataset of 8,641 pairs. Figure **??** reports on the distribution of ratios of web page versions in the consolidated WAYBACK dataset.

Mutation Frequency. Figure **??** further analyzes the above dataset by reporting on the ratio of mutations by category occurring between two versions of a web page.

As expected, the mutation ratio increases with the gap between the versions. Most importantly, the average mutation ratio in the WAYBACK dataset is 60%, i.e. in average, 60% of the nodes have mutated between two versions D and D' from the Wayback dataset. This mutation ratio is significantly higher than the average amount of mutations in the MUTATION dataset: 20%. This difference is probably an important factor justifying the differences of accuracy measured on both datasets. Comparing versions with large gaps is practical because it ensures there will be dif-

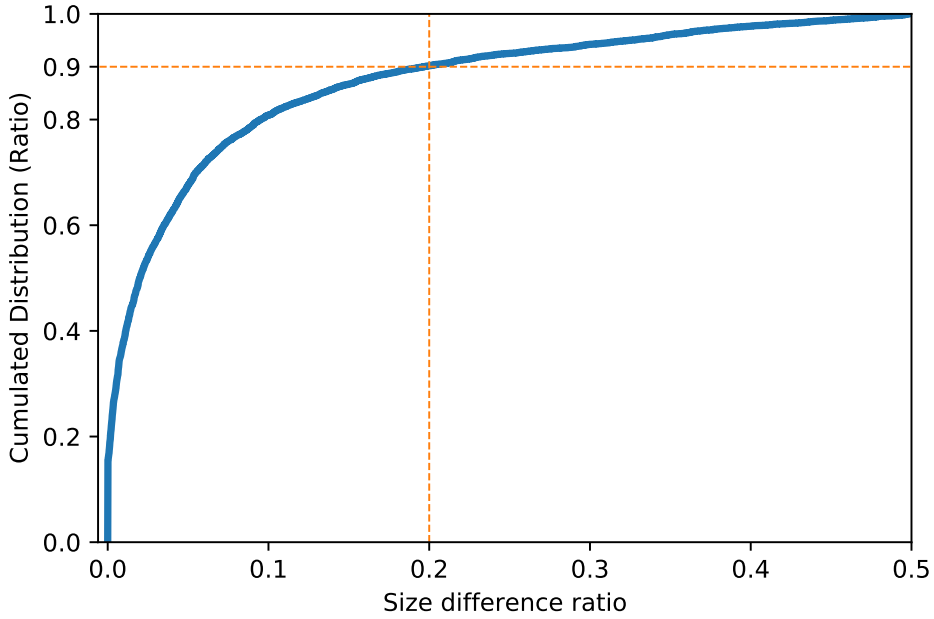


Figure 3.13: Cumulative Distribution of ratios between two versions of web pages. The orange dotted lines show the threshold used in this experiment

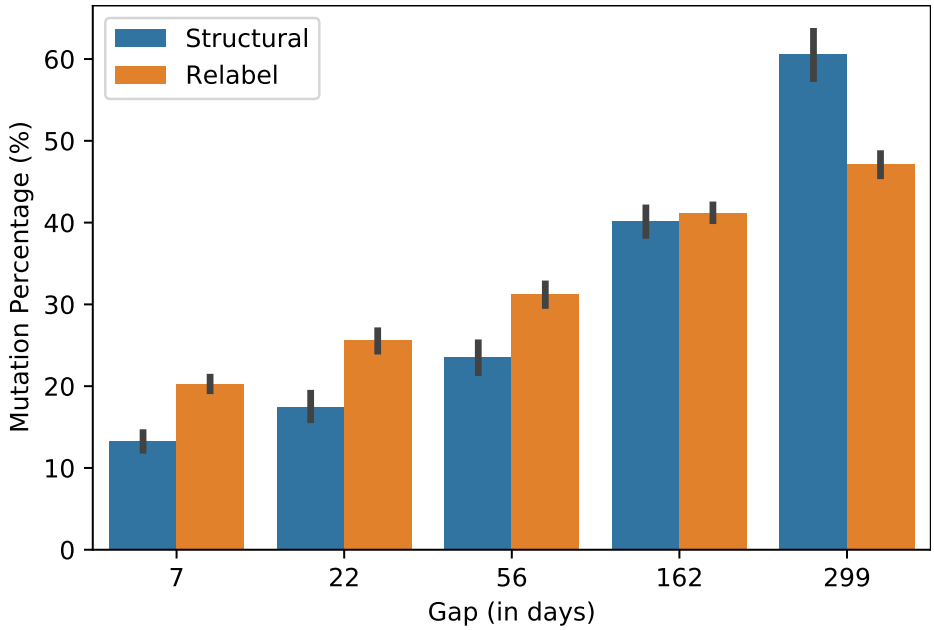


Figure 3.14: Mutation ratio between two WAYBACK snapshots depending on gap duration.

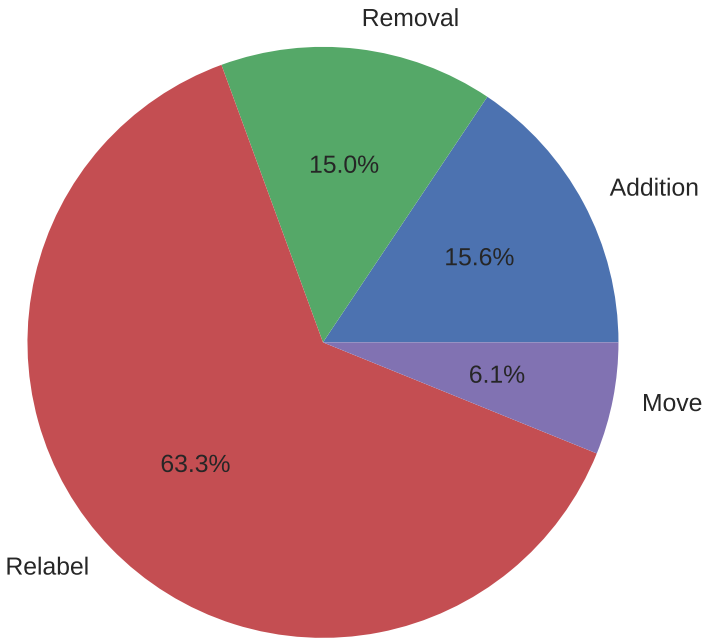


Figure 3.15: Distribution of mutation labels in the WAYBACK dataset.

ferences between the versions. In addition, it provides interesting insights about the frequency of changes on popular web pages. However, in the context of web testing, the mutation ratio between two consecutive versions is unlikely to reach 60%, in particular when adopting test-driven developments.

Mutation Labels. Figure **??** further describes the distribution of mutation labels in the WAYBACK dataset. The figure highlights that relabel is the most common mutation, while move is quite rare. This result can be explained by the following observations: 1. when a subtree is moved, only one move is accounted even if the visual change may appear as important, and 2. the SFTM algorithm is particularly robust to relabels, which could also be a factor explaining the observed ratio.

3.7.3 Repair Time Evaluation

In Figure **??**, we compare the computation times of ERRATUM and WATER. The results have been obtained by running both algorithms on the same server containing 252 GB of RAM and an Intel(R) Xeon(R) CPU E5-2660 v3 @ 2.60 GHz.

ERRATUM works differently than WATER: while WATER matches one single element at a time, ERRATUM matches all elements at once. One can observe that

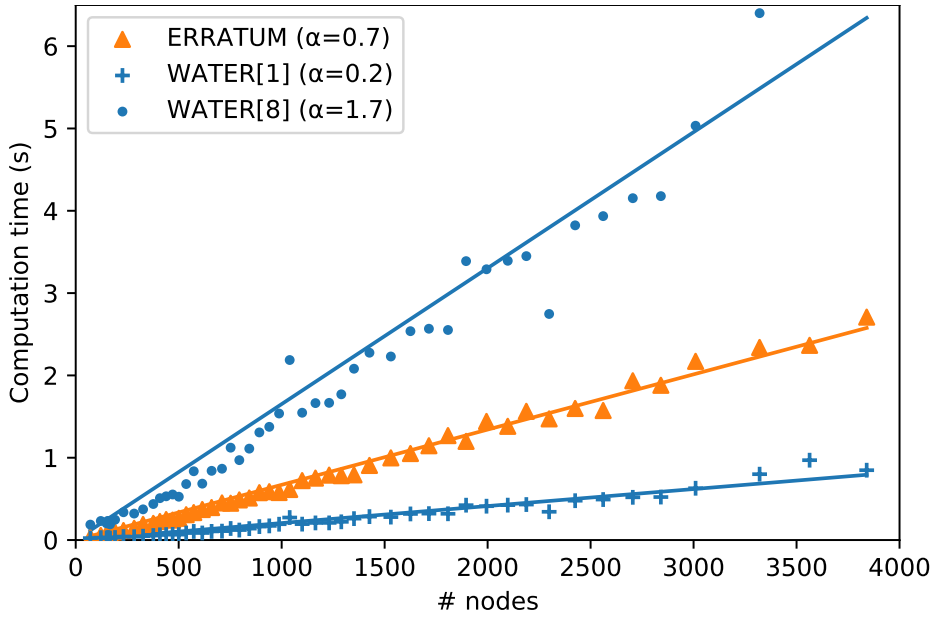


Figure 3.16: Repair time evolution according to DOM size.

WATER is thus faster at locating a single element than ERRATUM is at locating all elements. However, when the number of locators to repair grows, the computation time of WATER evolves proportionally, while the computation time of ERRATUM remains the same.

More specifically, we compare the evolution of the performance coefficient (α) when increasing the number of locators to repair in a web page. Figure **??** plots these coefficients for ERRATUM and WATER, so that we can establish that ERRATUM becomes more efficient than WATER as soon as there are more than 3 locators to repair in a web page.

3.7.4 Threats to Validity

As described in Section **??**, the WAYBACK dataset does not include a ground truth (perfect matching). This is why we had to manually label a representative sample of the matchings obtained on this dataset, which might have introduced some bias. To mitigate this bias, we recommended systematic and consistent decisions to label the data (cf. Section **??**).

All our experiments with ERRATUM adopt the default FTM parameters, as recommended by [**?**]. Nonetheless, a thorough parameter sensitivity study would probably result in further improving the accuracy of ERRATUM. Given the results we obtain on a wide diversity of web pages evolutions, we believe that this parameter tuning

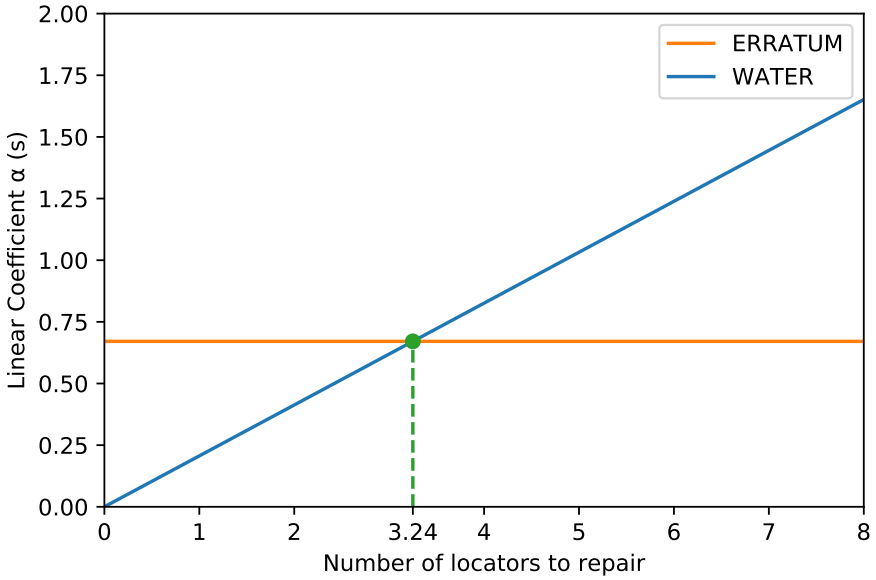


Figure 3.17: Performances of ERRATUM and WATER.

would only positively and marginally impact the accuracy of ERRATUM.

In terms of repair time, we discussed the absolute value of repair time for both solutions. However, these values highly depend on the way each tool was implemented and the machines on which the simulations were run. To limit this bias, both solutions were executed on the same Node.js runtime version deployed in the same environment to ensure a proper comparison.

3.8 Applying ERRATUM

We have studied how ERRATUM can help in solving the existing locator repair problem, which is a common problem in web automation scripts. In this section, we envision a more interactive development process made possible by ERRATUM.

When developing a web automation script, a developer typically opens the page under test in the browser, visually locates the element to interact with and then encodes (or generates) a locator for this element. The locator is then used in the web automation script to select the target element and interact with it. Based on the results achieved by ERRATUM, the perspectives for this work include a new layer of abstraction to the target selection. In this new layer, web automation scripts no longer need to explicitly locate elements on the page directly, but only locate elements using a back-end service H:

1. each web page D under automation is registered in H,

2. for each registered web page D, H exposes a visual interface allowing the developer to visually select an element e and generate a unique identifier e_{id} (e.g UUID),
3. in the web automation script, instead of using a manually encoded (or generated) XPath or CSS locator to select the target element e, the developer sends (D, e_{id}) to H's API, which returns an absolute XPath selecting the target element e,
4. when a web page D registered in H evolves into a new version D', ERRATUM is automatically used to relocate all registered elements e_{id} in D with their matching elements in D',
5. whenever ERRATUM fails (or lacks confidence) to relocate an element, the developer is notified and invited to visually relocate the broken locator.

This approach differs from the test repair approach described in the original WATER article. In the test repair approach, the locator repair is triggered by the failure of one of the test scripts. Once such a test script fails, the test repair solution attempts to determine the cause of the breakage and if it is a locator, repair the locator. The approach we suggest in this section does not include the analysis of any automation script, as locators are updated whenever the page changes.

In many cases, the locator breakage occurs silently (the locator is mismatched and the consequences happen only later in the test script) [?]. In these situations, it is harder to locate the origin of the breakage from the test script. The silent breakage problem happens because when using xPath locators to relocate e in D', the xPath query either succeed or fail. There is no indication on the confidence of the relocation that would help to detect a mismatch. On the opposite, Using ERRATUM, every individual match between two elements e and e' has an associated cost $s(e, e')$ that can be used as a confidence level to avoid mismatch.

In addition to the obvious gain in time that having a visual-based breakage and repair solution provides, the ERRATUM-based notification process described above would thus help to detect possible breakage before the scripts are even run, thus diminishing the chances for a "silent breakage" to occur.

3.9 Conclusion

In this article, we considered the situation in which the evolution of a web page causes one of its associated automation scripts to break. In the domain of automated web testing, this situation accounts for 74% of test breakages, according to past studies [?].

Our analysis of the state-of-the-art approaches on this topic contributed to formalize the key steps involved in preventing or fixing such a kind of test breakage.

While existing solutions to the locator repair problem treats broken locators individually, we rather propose to apply a holistic approach to the problem, by leveraging an efficient tree matching algorithm. This tree matching approach thus allows ERRATUM, our solution to repair all broken locators by mapping all the elements contained in an original page, to accurately relocate each of them in its new version at once. To assess ERRATUM, we created and shared the first reproducible, large-scale datasets of web page locators, combining synthetic and real instances,[4] which has been incorporated in a comprehensive benchmark of ERRATUM and WATER, a state-of-the-art competitor.[5] Our in-depth evaluation highlights that ERRATUM outperforms WATER, both in accuracy—by fixing twice more broken than WATER—and performances—by providing faster computation time than WATER when repairing more than 3 locators in a web test script.

Finally, we worked with the development team of a widely used open source test framework called Cerberus [6] to integrate [7] the Erratum approach into the test creation part of the software.

[4]Dataset available from `https://zenodo.org/record/3800130#.XrQb02gzY20`
[5]Benchmark available from `https://zenodo.org/record/3817617#.XrWdqGgzaoQ`
[6]https://cerberus-testing.com/
[7]https://github.com/cerberustesting/cerberus-source/commit/0a70d4cc0d70a797901652fd2b97d501bb7fa511

[sigconf,authordraft]acmart

[english]babel [utf8x]inputenc [T1]fontenc amsthm paralist algorithm algpseudocode amsmath graphicx [colorinlistoftodos]todonotes hyperref listings xparse [super]nth caption listing xcolor listings subcaption

Definition[section]

[1]// 1

A clear and well-documented LaTeX document is presented as an article formatted for publication by ACM in a conference proceedings or journal publication. Based on the "acmart" document class, this article presents and explains many of the common variations, as well as many of the formatting elements an author may use in the preparation of the documentation of their work.

datasets, neural networks, gaze detection, text tagging

APPSTRACT: Inferring Application-wide Locators for the Web

May 25, 2022

3.10 Introduction

Web applications are often perceived by end-users as online systems exposing a limited set of views, concepts, and related actions. From a user perspective, an online shop—like Amazon—mainly offers to search and list products, while clicking on a given product brings us to its details view. From a machine perspective, like a bot, navigating the same web application will be perceived as crawling thousands of unique pages, exposing unrelated content and seemingly unique actions.

This inability for a machine to understand the navigation model of a web application, like a user intuitively does, makes it very hard to automate interactions with web applications. Typically, research topics benefiting from more intuitive navigation include: *(a) web data mining* to automatically extract, de-noise, and structure data from web pages, *(b) web testing* to generate, maintain, repair, and augment tests on web applications, and *(c) web analytics* to report on how users interact with the web application.

While there is a considerable amount of existing literature, notably in the fields of data mining [?, ?, ?, ?, ?, ?], most of existing works are highly specific and do not provide adequate insights to build an application-wide understanding of a navigation model.

In particular, state-of-the-art approaches focus on either extracting data within a page (so-called, *intra-page* extraction) or across two pages (*inter-page* extractions).

For example, Miao *et al.* [?] clusters full tag paths—*i.e.*, `/html/body/div[2]/div[1]/span`—to detect repeating occurrences of a template within a page. This allows their approach to extract what are usually called *records* in a web page (*e.g.*, a single product card in a product list). This is an example of what we categorize as *intra-page abstraction*.

Inter-page abstraction solutions usually try to detect templates of a webpage or to learn a wrapper by studying two different instances of the same template. That is what is done by the state-of-the-art algorithms, like EXALG [?] and RoadRunner [?]. However, none of these approaches provide any insight to take into account the intra-page variability.

In this paper, we thus propose an unsupervised approach to infer the navigation model of a whole web application that we call an *appstraction*. The *appstraction* of a web application aims at delivering actionable insights: it allows a machine to understand application states (*e.g.*, product pages, blog page), as well as elements within states (*e.g.*, product title, price), hence guiding the information extraction

process, as well as identifying relevant navigation actions. The *appstraction* process aims to abstract away the natural variability of web pages into a canonical model built as a compact tree of *template pages* and *template elements*. Our unsupervised approach assumes that even web applications with billions of pages will build on a limited set of template pages, thus making it possible to infer these generative templates from a dataset of visited pages. To achieve this, our approach—named APPSTRACT—builds on three stages: 1. *web page clustering* to group instances of related web pages, 2. *intra-page abstraction* to extract repeating patterns within each cluster of pages, and 3. *inter-page abstraction* to capture repeating patterns across clusters.

We empirically demonstrate that our *appstraction* succeeds to generate application-wide locators that can be used to support semantic guidance across multiple pages of any web application. **MORE TO BE SAID ON THE DEMONSTRATION**.

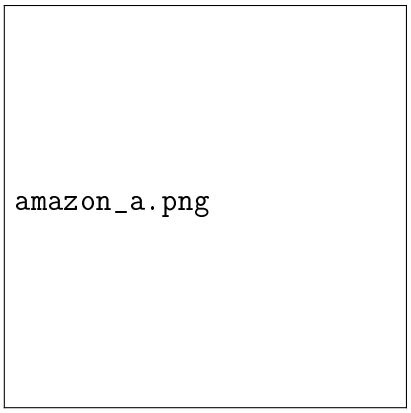
The remainder of this paper is therefore organized as follows. Section **??** introduces the required background and related works in this area. Section **??** presents the design and implementation of APPSTRACT, while Section **??** reports on an evaluation of the perceived accuracy of APPSTRACT. Finally, Section **??** concludes.

3.11 Background & Related Works

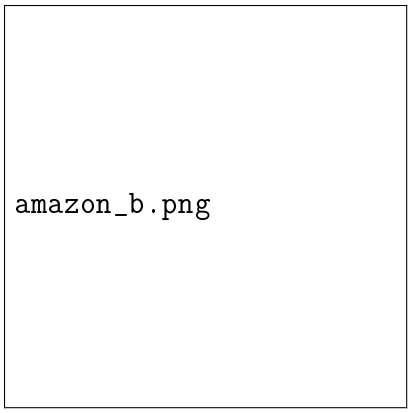
In this section, we present several studies across different research fields that offer partial solutions to the general problem of web application abstraction inference. Throughout this paper, we consider a subset of the Amazon web application as a running example. In particular, we focus on two related page templates: the product details and product list pages (cf. Figure **??**).

Data Extraction

The role of a web page is mainly to *present* a subset of the information it has access to in a certain way. All data extraction solutions attempt to separate the information from its presentation—*i.e.*, the template. The process of data extraction is thus a form of abstraction of an application: instead of viewing the different pages as a multitude of unrelated blobs of HTML, the application is seen as a limited set of templates that presents various information in a consistent way. The process of information extraction is thus closely related to our *appstraction* objectives and our approach is highly inspired by the ideas behind data extraction.



(a) Sample page from cluster *list*.



(b) Sample page from cluster *product*.

Figure 3.18: Screenshots illustrating some elements $\hat{e}_\lambda$ included in the template pages $\hat{p}_{list}$ and $\hat{p}_{product}$ of 2 distinct clusters.

Following the classification made by a survey on data extraction [**?**], most existing literature on data extraction can be classified according to the levels of supervision needed to extract the data:

- *(semi-)supervised approaches* where the user inputs more or less detailed directions to describe to how to extract data [**?**, **?**, **?**, **?**, **?**, **?**, **?**],
- *unsupervised approaches* where data is automatically extracted by analysing recurring patterns [**?**, **?**, **?**, **?**, **?**].

Supervised approaches are usually qualified as "wrapper-based". The idea of wrapper-based data extraction is to consider the set of web pages containing the data to extract as an unstructured (or semi-structured) database and build a query language (and its query engine) to query the desired data.

In this paper, we focus solely on *unsupervised approaches*. These approaches rely on the assumption that, even though there is a lot of different pieces of information exposed by an application, the same type of information will always be structured in the same way. For example, on the product list page of an online shop, each product will be structured in a similar fashion (*e.g.*, product name, price, description). In this context, we distinguish two families of unsupervised approaches: *intra-page* and *inter-page* data extraction. One should note that *intra-page* data extraction is usually referred as *record extraction* [**?**, **?**].

Intra-page data extraction refers to the extraction of data within a page. For example, in the case of the Amazon page presented in Figure **??**, an intra-page data extraction solution, such as MDR [**?**], relies on the topology of the DOM tree and

string matching to detect data regions $\hat{e}_{product}$ within a single page. As it is always the case when attempting to extract data in an unsupervised way, MDR can only detect data regions if at least two of these regions are present on the page. This is a necessary condition since all unsupervised algorithm must rely on some kind of pattern discovery to detect data regions.

Inter-page data extraction refers to the extraction of data across several pages. In particular, all existing algorithms apply to pages that are assumed to belong to the same template (*e.g.*, two Amazon product pages). The idea is then to use the similarity between the two pages to understand what is common and what has changed between the two pages. The parts that changed are then assumed to be data, while the common part has to do with its presentation (*template*). In order to compare these pages, one solution [**?**] uses a modified version of the most popular general tree matching solution: *tree edit distance*. In the inter-page abstraction part of our approach, we also use a tree matching solution but not the same one.

The challenge of data extraction presented above is very similar to that of our *appstraction* objective. However, it differs in a few important points:

1. while data extraction focuses on extracting data, we try to abstract any kind of variability. For example, in Figure **??**, a data extraction solution should not attempt to extract the $\hat{e}_{filter}$ template elements, as they do not represent data.

2. several data extraction studies are focused on how to infer a schema of the extracted data (*e.g.*, relational schema [**?**, **?**])), which we are not interested to do in the context of *appstraction*, and

3. most importantly, to the best of our knowledge, no previous work has attempted to extract data throughout a whole web application (combining both inter-page and inter-page abstraction).

3.11.1 Web Testing

Web testing covers a large body of research that encompasses various themes, such as test generation, test coverage, or test robustness. We claim that, essentially, the main challenge of web testing comes from the lack of abstract understanding of the application under study and most works in this field have to resort to ingenious ideas to compensate for this lack of abstract model.

MAYBE REFER TO THE PAPER WITH XAVIER ON EXPLORATORY TESTING AS THEY BUILD MANUALLY SUCH AN abstraction THROUGH

THEIR DSL.

Test Robustness. One of the main challenges of web testing encountered by the industry is test breakage: tests written for a given version of a web application break when the application evolves. For example, let us consider a testing script that applies to a web page D. One part of the script instructs to click on a given button e in the page D. To locate e, scripts usually use xPath or CSS selector to build what is called a *locator* l_e. Breakage can then happen when a new version D' of the page D is published. Most of these breakage are locator based [?]: the locator l_e that successfully located e on D does not locate the matching element e' in D'. To solve this problem, some attempt to automatically generate more robust locators relying on the structure to the tree [?, ?, ?, ?] while other works offered solutions to repair broken locators [?, ?, ?].

In particular, The ERRATUM approach [?] achieves high accuracy specifically thanks to a more holistic stance: instead of looking at individual locators independantly, the approach considers the page as a whole using a similar technique to the *inter-page abstraction* part of our *appstraction*.

The "locator" problem is a more specific instance of the web application abstraction: the breakage happen because when a machine sees a multitude of different pages (*e.g.*, different versions), a human perceives a single page template with slight differences and thus comes to expect the locators on D should naturally work on D'.

In the present work, we go beyond ERRATUM [?] on the generalization scale and create "locators" that are valid through all pages of a web application.

Test Generation ?

3.11.2 Web Analytics

We know of no work that allows to create a full model of a web application.

2021 Modeling Web Browsing Behavior across Tabs and Websites with Tracking and Prediction on the Client Side [?]

- Web behavior analysis using clickstream
- Novelty: Client Side only / Multi-tab analysis
- End results: Classify navigation types / Predict user next click
- How: url encoding - weird RNN fed with actions and timestamps

- Eval: Give missions to users on different websites (132 action paths / 19 test)
- Results: classification -> 100%, prediction -> not amazing

What did they do? Understanding clickstreams with the WebQuilt visualization system [?]

3.12 Appstract

3.12.1 Abstracting a Web Application

This section starts by defining the notions of *application* and *application abstraction* we assume, before discussing the challenges of inferring such defined abstractions.

Definition 3.12.1 (*Application).* *A web application A is a set of web pages $\{p \in A\}$, where every page p is captured by a* Document Object Model *(DOM).*

The number of web pages $\|A\|$ can be very high and keep rising with time (*e.g.*, Amazon products or blog posts). In addition, we consider that any mutation of a web page (even if the URL does not change) is considered as a new page. Obviously, from a human perspective, a web application is much more than a collection of pages (*e.g.*, the pages are linked, the application has different features usable by different users), but we choose to take the perspective of a machine, for which an application is perceived as a set of visited web pages. As a DOM of a page p is a tree, we represent it as a tuple $\langle N(p), par\rangle$ where $N(p)$ is the set of elements (or nodes) in p and $par : N(p) \rightarrow N(p)$ is the parent function that associates a DOM node with its parent. To lighten the notations, we write $e \in p$ to describe an element e in a page p, instead of writing $e \in N(p)$.

Definition 3.12.2 (*Application abstraction).* *The abstraction of a web application A is a tuple $\langle \hat{A}, T_{\hat{A}}\rangle$ where* i) *$\hat{A}$ is the set of template pages and each page $\hat{p} \in \hat{A}$ contains template elements $\hat{e} \in \hat{p}$ and* ii)*:*

$$\begin{aligned} \forall p \in A, T_{\hat{A}}(p) = \langle \hat{p}, T_{\hat{p}}\rangle \\ and\ \forall e \in p, T_{\hat{p}}(e) = \langle \hat{e}\rangle \end{aligned} \tag{3.3}$$

In other words, the abstraction of a web application A is a set of template pages (*e.g.*, product page, list page) and a function that allows to map any page from A to its corresponding template, and every element within the page to the corresponding element in the template page. This mapping is completed by two functions:

1. $T_{\hat{A}} : A \to \hat{A}$ is the template function that takes any page $p \in A$ and returns the matched *template page* $\hat{p} \in \hat{A}$, and

2. $T_{\hat{p}} : N(p) \to N(\hat{p})$ is a function that takes any element $e \in p$ and returns the matched element in the template $\hat{e} \in \hat{p}$.

Additionally, we also use:

- $T^{-1}_{\hat{A}}(\hat{p}) \subset A$ as the set of pages $p \in A$, such that $T_{\hat{A}}(p) = \hat{p}$, and
- $T^{-1}_{\hat{p}}$ as the inverse function of $T_{\hat{p}}$.

These notations allow us to easily refer to the instance pages/elements of a given template page/element.

Figure **??** depicts a theoretical application of the appstraction to 2 web pages crawled from Amazon. For each related clusters of similar web pages, the $T_{\hat{A}}$ function will return either template page $\hat{p}_{list}$ or $\hat{p}_{product}$. Then, for each clustered web page, each web element can be mapped to its corresponding template element (*e.g.*, $\hat{e}_{title}$, $\hat{e}_{product}$) using the $T_{\hat{p}}$ function.

Please note that our *appstraction* process does not intend to label template pages or elements. Thus, our references to $\hat{p}_{list}$ or $\hat{e}_{product}$ should be understood as a *global unique identifier* (GUID) capturing a class of pages and elements within and across clusters.

3.12.2 Building an Abstraction

Figure 3.19: Overview of the two stages of the *appstraction* process: *learning* and *prediction*. The identifier map of p noted T_p associates an identifier to each element of p. The identifiers are first randomly generated, then merged to existing maps during inter-page abstraction. The prediction phase produces an identifier map $T_{\hat{p}}$ in which each original element of p is associated to an application-wide identifier.

To abstract the web page p into its *appstraction* A_p, we operate at two levels:

1. *intra-page abstraction* to extract repeating patterns of elements within a cluster of web pages, and

2. *inter-page abstraction* to group repeating patterns of elements across template pages.

This bi-level abstraction process combines two steps: *learning* and *prediction*, as depicted in Figure **??**. In this section, we first overview the abstraction process by considering intra- and inter-page abstractions as black boxes. In the following sections, we provide an in-depth description of individual level abstraction techniques.

Learning The objective of the learning phase is to build a model of the application that will deliver accurate predictions. The upper part of Figure **??** describes how intra- and inter-page abstractions are used to build this model.

Before any abstraction, the application is perceived as a set of related pages. The first step is thus to organise these pages into clusters, each cluster representing a template page (*e.g.*, product page). In order to do this, any clustering algorithm could be used and, in this paper, we consider the clustering part as out of scope as we focus on the actual abstraction process. The learning approach thus starts with the user picking one example page per template.

Each template page goes through intra-page abstraction. The intra-page abstraction takes a DOM tree p as input and returns a *template* as output. A *template* is a tuple $t = \langle \hat{p}, T_p \rangle$ where T_p is the identifier map that maps every element from the original tree p to a global identifier. After the intra-page abstraction, $\hat{p}$ typically contains much less elements than p since all repeated patterns have been abstracted away. The T_p map is a way to keep track of the elements in the original page p after abstraction. For example, in Figure **??** illustrating intra-page abstraction, the identifier map would contain 11 entries, one for each original element in p and 5 distinct values corresponding to the five distinct nodes of $\hat{p}$.

In practice, the values of T_p are *Global Unique Identifiers* (GUID). Before starting the abstraction process, we transform the page p into a *template* tuple, where the identifier mapping maps every node to a randomly generated GUID. Each abstraction step will thus transform the input tree and update the associated identifier map.

Once each page has been abstracted at the intra-page granularity, we build a model allowing to achieve two aspects of inter-page abstraction:

- *Template abstraction:* same elements within different instances of a template must have the same identifiers (*e.g.*, title of a product on different product pages)
- *Cross-template abstraction:* same elements within different pages (regardless of templates) should have the same identifiers. (*e.g.*, menu link that appears on all template pages)

To achieve cross-template abstraction, we arbitrarily select one template to represent the whole application ($\langle \hat{p}_a, T_{p_a} \rangle$ in Figure **??**) we call this template the *mother template*. All other template pages are then inter-page abstracted against the mother template. The inter-page abstraction function takes two templates: a reference template and a query template. It then returns one template in which the values of the identifier map have been updated to match the reference template.

The final model is obtained after applying intra-page abstraction between the mother template and all other pages. The final model is simply a set of n templates where each template is a tuple $\langle \hat{p}, T_p \rangle$.

Prediction During the prediction phase described in the bottom part of Figure **??**, the user sends a previously unseen page and APPSTRACT returns the mapping between each element e of the page and the *id* of the associated template element $\hat{e}$.

To do so, APPSTRACT first applies intra-page abstraction to create an abstracted DOM tree $\hat{p}$, then applies the tree matching algorithm at the core of the inter-page abstraction between $\hat{p}$ and all templates in the model. The template page from the model that matches best with $\hat{p}$ is retained (*e.g.*, $\hat{p_b}$ in Figure **??**).

The matching between $\hat{p}$ and the corresponding template is then used to compute the final mapping $T_{\hat{p}}$.

Overall, the user sent a page and received the mapping between each element of the page and the corresponding template element in the corresponding template page.

In the following sections, we describe both intra-page abstraction and inter-page abstraction in more details.

3.12.3 Intra-Page Abstraction

Figure 3.20: Illustration of Intra-Page abstraction. DOM leaves at the end of repeating branches are tagged and then recursively merged.

Intra-page abstraction relies on the detection of repeating patterns within a page.

Building the intra-page abstraction corresponds to creating the T_e function defined above. The intra-page abstraction deals with a single page: given an input page p, we want to build a template page $\hat{p}$ and a function T_e that maps all the elements $e \in p$ to their corresponding elements $\hat{e} \in \hat{p}$ Overall, the Intra-Page abstraction is done in three steps:

1. *Leaf clustering* clusters repeating DOM leaves together,

2. *Node tagging* propagates information about leaf-clusters in all ancestor nodes to prepare for the final step,

3. *Recursive leaf-group merging* builds the intra-page abstraction by recursively merging branches.

Leaf Grouping

In the first step of intra-page abstraction, we attempt to identify groups of leaf nodes that present data of the same type. To do so, we: 1. build all root-to-leaf paths from the DOM tree, 2. group all same paths together, and 3. filter out groups of leaves containing less than a fixed threshold k of elements. At the end of this phase, we have a set of leaf groups (LG), each containing at least k elements.

The *root-to-leaf* path of a node is the formatted sequence of tags from the root to the node. For example, on the web page from Figure **??**, the root-to-leaf path of the element containing the text *price2* is `//html/body/div/span/`.

```
<html>
  <head> <!-- header --> </head>
  <body>
      <div> <!-- content --> </div>
      <div>
          <a href="...">Item1</a>
          <span>price1</span>
      </div>
      <div>
          <a href="...">Item2</a>
          <span>price2</span>
      </div>
  </body>
</html>
```

Figure 3.21: Web page example to illustrate intra-page abstraction with no nested records

To find leaf-groups, we extract all root-to-leaf paths from the DOM tree, group the same ones together and only keep the groups that contain more than a fixed threshold k of elements. In example **??**, assuming $k = 2$, it means we have two groups:

1. Leaf group 1 (`//html/body/div/span/`) containing `price1` and `price2`, and
2. Leaf group 2 (`//html/body/div/a/`) containing `Item1` and `Item2`.

In our evaluation however, this threshold is set to $k = 4$.

Limits Our grouping method may fail to cluster items correctly in two situation:

- *Over-abstraction*: Items can be clustered together even if they are not of the same type, or
- *Sub-abstraction*: Items can be put in different clusters even though a human would put them in the same cluster.

To a certain extent, *sub-abstraction* can be compensated afterwards. For example, when extracting information, it is common for websites to structure data differently according to the type of exposed products (*e.g.*, regular products or "sponsored" products). In this case, using the approach we presented, the two types of products will be classified in different clusters, meaning that if the data is extracted as a table, the prices will be spread in two different columns that the user will be able to easily merge.

The cases of *over-abstraction* are more problematic since it will mean that the associated data has been lost when abstracting the page. **EST-CE QUE ÇA FAIT PARTIE DE L'ÉVALUATION POUR QUANTIFIER À QUEL POINT ÇA POLLUE LES RESULTATS? -> oui ca fera partie integrante de l'evaluation**

Node Tagging

At this stage, we have a list of all leaf-groups ($LG_1, LG_2...$). In order to be able to recursively merge all leaf-groups, we propagate the information that a leaf belongs to a certain group to all ancestors of the leaf using the node-tagging algorithm **??**.

[1] createTags$LGs = [LG_1, LG_2...]$ tagBranch$depth$, LG, e $tag \leftarrow \langle LG, depth \rangle$ $e.tags[tag] += 1$ Inc or init tag count of node e tagBranch($depth + 1$, LG, $e.parent$) **forall** $LG \in LGs$ **do**

end

$e_{leaf} \in LG$ tagBranch(0, LG, e_{leaf})

Algorithm 3: Intra-Page abstraction: Node Tagging

Algorithm **??** iterates through all the leaves that belong to a leaf group LG and for each leaf, it recursively tags all the ancestors of the leaf. The *tag* of an element e is the tuple $\langle LG, depth \rangle$ where LG is a leaf-group and *depth* is the number of nodes between e and the leaf from its offspring that belongs to LG. The tag is used to identify nodes that should be merged together.

When the algorithm ends, each node e of the DOM tree contains a map *tags* whose keys are tags and values are integers count. Figure **??** shows an example result of the node-tagging algorithm on a simple DOM tree.

Figure 3.22: Output of the node-tagging algorithm. Each node is assigned a *tags* map keeping track of the number of leaf-groups in its offspring.

An important implementation detail: the above algorithm assumes that the *tag* tuple's hash code will be generated from the value of its components (and not the object's reference)—*i.e.*, two *tag* tuples containing the same leaf group LG and the same *depth* should have the same hash code when inserted into the *e.tags* map (even though the tags will have different addresses in memory since they were created at a different stage of the algorithm).

Recursive Branch Merging

Overview The last step of the intra-page abstraction consists in merging all required nodes such that the resulting DOM contains only one node instance of each leaf group. Figure **??** shows an example of the result obtained after this last step of the intra-page abstraction. In the final abstract tree of Figure**??**, the red dotted edge connecting the template element $\hat{e}$ to its parent indicates that $\hat{e}$ has a special relationship with its parent, in this case: a 1-to-many relationship.

The algorithm must be recursive because it is possible (and likely) that some leaf groups will have to be merged at different levels of the tree. Figure **??** illustrates this use case: e_{a1} and e_{a2} must be merged first before their ancestor e_{b1} can merge with e_{b2}.

(a) Nested records (b) Optional elements

Figure 3.23: Illustrating two important cases our merging algorithm must cover: *nested records* and *optional elements*.

The second case illustrated in Figure **??** occurs when merging two nodes even though not all their respective children are merged. In this case, the inferred abstract node $\hat{e}_1$ is said to have an *optional* relationship with its parent template element—*i.e.*, it only exists as a child node in certain instances of $\hat{e}_1$. In general, a template element $\hat{e}$ can have many types of relationship with its parent. In this paper, we distinguish two:

- A *zero-to-many relationship* when at least one element from $T_e^{-1}(par(\hat{e})) = par(e_1), par(e_2)...$ has no child element that is an instance of $\hat{e}$ and at least one other element has more than 1,

- An *optional (or zero-to-one) relationship* when at least one element from $T_e^{-1}(par(\hat{e})) = par(e_1), par(e_2)...$ has no child element that is an instance of $\hat{e}$ and at least one other element has more than 1.

Algorithm Algorithm **??** gives a high level view of the algorithm. The function *abstractTree* recursively traverses the tree. For each node e considered, it checks if e is already abstract using the *isAbstract* function (line **??**). A node is abstract if none of its children need to be merged. If e is already abstract then it is returned, otherwise, we:

1. recursively call the *abstractTree* function on all children of e,
2. group all abstract children by group of tags using *assocGroup* (see paragraph **??**), and
3. merge each group of children using *mergeGroup* (see paragraph **??**)

[1] abstractTree$e$: *Element* **if** *isAbstract(e)* **then**
end
alg:abstractTree:isAbstract **return** e **else**
end
children $\leftarrow$ e.children.map(abstractTree) children $\leftarrow$
assocGroup(children).map(mergeGroup) $\hat{e} \leftarrow$ { e with e.children = children }
return $\hat{e}$

Algorithm 4: Intra-page abstraction: recursive merge

We describe in details the three functions used in algorithm **??**, namely: 1. *isAbstract*, 2. *assocGroup*, and 3. *mergeGroup*.

isAbstract The function *isAbstract* checks if an element e is already abstract. An element e is abstract if none of its offsprings need to be merged. This information is obtained using the tags computed in previous steps (see Section **??**).

Each tag *tag* associated to a node e is associated to its tag count $|tag|$. Using the tag counts, we can then detect if e is abstract: a node e is abstract iff all tag counts are equal to 1. In this case, it means that no node in the offspring will have to be merged. For example, in Figure **??**, all nodes are abstract except e_0 since two tags in $e_0.tags$ have a tag count of 2. In practice, it means that some children of e_0 (in this case e_1 and e_2) will have to be merged.

assocGroup The function *assocGroup* (for associative grouping) groups all the nodes that need to be merged together. In order to know if two nodes need to be

merged, we look at their associated *tags* map. The merging condition is simple: two nodes should be merged iff they share one same tag. Below is an example of input/output pair of the *assocGroup* function:

```
Input:
    A has t1, t2
    B has t2, t3
    C has t3
    D has t4
    E has t4

Output:
    [A, B, C] -> [t1, t2, t3]
    [D, E] -> [t4]
```

In our case, the letters A to E are nodes and $t1$ to $t4$ are tags.

mergeGroup ?? The function *mergeGroup* is the core of the recursive merging algorithm. As an input, it takes a list of abstract nodes—*i.e.*, abstract subtrees—that need to be merged and the set of common tags between the groups of nodes. The output is the abstract node $\hat{e}$. The inputs come from the output of the function *assocGroup*, described above, and the abstract node output replaces all children that were merged in the tree, as shown in the *abstractTree* algorithm **??**

The function *mergeGroup* reduces the group list taken as input by repeatedly applying the function *mergeAbstractTrees*. Algorithm **??** describes how the function merges two abstract trees into one.

[1] mergeAbstractTrees$\hat{e}_1, \hat{e}_2$ Group pairs of children containing the same tags pairs, orphans $\leftarrow$ groupPairs($\hat{e}_1$, $\hat{e}_2$) mergedChildren $\leftarrow$ pairs.map(mergeAbstractTrees) **for** *e in orphans* **do**
end
e.rel $\leftarrow$ relType.Optional
$\hat{e} \leftarrow$ new Node() $\hat{e}$.tag = $\hat{e}$.tag $\hat{e}$.attrs = mergeAttrs($\hat{e}_1$.attrs, $\hat{e}_2$.attrs) $\hat{e}$.rel = mergeRel($\hat{e}_1$.rel, $\hat{e}_1$.rel) $\hat{e}$.children = mergedChildren + orphans $\hat{e}$.tags = $\{\hat{e}_1 \cup \hat{e}_2$ with tag counts set to 1$\}$ **return** $\hat{e}$

Algorithm 5: Intra-page abstraction: merge two abstract trees

Before diving into the details of the algorithm, it is important to highlight that the inputs of the function *mergeAbstractTrees* are already abstract. At this stage of the algorithm, we are assured that along the whole branch starting at the root of

both nodes sent as parameters, there is never two children belonging to the same cluster—*i.e.*, that needs to be merged. It means that the algorithm sole purpose is to recursively merge the two trees between them (and not within).

The function starts by grouping the children of the two nodes into pairs and orphans. The function *groupPairs* returns two lists: the pairs of nodes that must be merged and the orphan nodes. The orphan nodes are the children in $e_{\hat{1/2}}$ that have no corresponding element to be merged with in $e_{\hat{2/1}}$, these elements are set as optional using the *rel* (as in relationship) property.

Before merging, we recursively call the function *mergeAbstractTrees* on each pair of nodes returned by *groupPairs*. Intuitively, the algorithm will stack the calls to *mergeAbstractTrees* until it reaches the leaves of the trees, then it will merge the groups of leaves and merge their ancestors as the function calls unstack.

Most of the steps described above help computing the *children* property of the abstract node returned by the *mergeAbstractTrees* function. Other properties of the nodes are also merged:

***rel*:** In case the relationships of the nodes to merge are different, they are merged using the following pattern matching:

```
mergeRelTypes :: (RelType,RelType)->RelType
mergeRelTypes (t1, t2)
    | (t1, t2) when t1 = t2 -> t1
    | (t1, Normal) -> t1
    | (_, ZeroToMany) -> ZeroToMany
    | (Optional, OneToMany) -> ZeroToMany
    | (t1, t2) -> mergeRelTypes (t2, t1)
```

***attrs*:** In case the attributes of the nodes to merge are different, they are merged. To merge the attributes, we select the attributes that are present in both nodes and merge their values using the *Longest Common Subsequence* (LCS) algorithm. For example, given three nodes having the following class values: *class* attribute: `"link nav-link"`, `"link active nav-link"` and `"link nav-link"`, the function *mergeAttrs* will return the following value: `"link nav-link"`.

Stack Diagram Since the algorithm has several levels of recursion, it may be hard to understand how a given tree will be abstracted.

In Figure **??** we describe the different steps of the algorithm. At each step, we show the output of the current function that is called. Below each tree, we show the

Figure 3.24: Key functions involved in the *abstractTree* algorithm. A directed arrow from function f to g indicates that f calls g.

Figure 3.25: An example application of the *abstractTree* function. For each figure, the stack of the current step is shown below.

current stack of functions called.

The functions we mention are all described earlier:

Figure **??** describes an example application of the *abstractTree* function. At each step, the current stack of functions is shown at the bottom. All functions shown have been described in the previous sections. Figure **??** summarizes all these functions and how they interact. For simplicity, however, the *mergeTwoAbstractTrees* is not explicitly mentioned, it is considered as part of the function *mergeGroup*.

Complexity

3.12.4 Inter-Page Abstraction

We described how to detect and abstract the repeating patterns contained within one page. In the second part of the APPSTRACT approach, we detect and abstract repeating patterns across pages of the web application.

The inter-page abstraction relies on tree-matching. A tree matching solution allows to match two web page DOM trees p and p'. The matching $M_{p,p'}$ obtained is a subset of $p \times p'$ such that each tuple $(e, e') \in M_{p,p'}$ represents the fact that the element $e \in p$ matches with the element $e' \in p'$ (e.g. e and e' contain the names of two different products on two different product pages p and p').

As described in section **??**, inter-page abstraction is used at both learning and prediction phases. In both cases, the inter-page abstraction can be described as a function that:

1. takes two templates as input: the reference template and a new template,
2. returns a template in which every template element $\hat{e}$ of the new template references its corresponding template element $\hat{e'}$ in the reference template (if a match was found).

Algorithm describes the inter-page abstraction process. The function role is to create a new identifier map T_{new} in which each element from p maps to the id of the matching element in p_{ref} (if there is any) or remains the same.

[1] inter$\langle p_{ref}, T_{ref}\rangle, \langle p, T\rangle$ $M \leftarrow$ tree-matching(p, p_{ref}) $T_{new} \leftarrow$ T.map((e, id) $\rightarrow T_{ref}[M[e]]$ if e in M else id) **return** $\langle p, T_{new}\rangle$

Algorithm 6: Inter-page abstraction

Figure **??** describes how the function *inter* is used for both learning and prediction:

1. during learning, we apply inter-page abstraction between the mother template and each of the other templates. This step allows to build a model of the application where elements that appear in all page templates will have the same id,
2. during prediction, inter-page abstraction is used to match all elements from an unseen page to the template elements of its matching template page.

3.13 Limits

Modeling a whole application is a highly ambitious task. We hope our work can help progress toward this goal, but we cannot claim that it already does. Indeed, our current work has several limits, mainly:

Template Topology Real-life templates may have a much more complex structure than the one we assume:

1. there can be a tree-like structure with deeply nested templates,
2. there could be graph-like template structure (*e.g.*, components).

Our current *appstraction* method do not allow to infer such template topologies.

Mother Template Selection During the learning stage, we choose the mother template arbitrarily among the existing template pages. This selection process assumes that all template pages have an equivalent amount of common parts.

3.14 Conclusion