SFTM: Fast Matching of Web Pages using Similarity-based Flexible Tree Matching

Sacha Brisset, Romain Rouvoy, Lionel Seinturier, Renaud Pawlak

SFTM: Fast Matching of Web Pages using Similarity-based Flexible Tree Matching

Sacha Brisset[a], Romain Rouvoy[a], Lionel Seinturier[a], Renaud Pawlak[b]

[a] *Univ. Lille / Inria, France*
[b] *Cinchéo, France*

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.

Keywords: Web, Tree matching, Information Extraction

1. Introduction

The success of Internet has led to the publication and the delivery of a deluge of structured content. Nowadays, web services and applications

Email addresses: romain.rouvoy@inria.fr (Romain Rouvoy), lionel.seinturier@inria.fr (Lionel Seinturier)

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 [20, 29, 30], web testing [3, 22], comparison of web service versions [7], web schema matching [8], and automatic re-organization of websites [13].

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) [24] 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 [17, 18, 19]. Despite these efforts, TED remains costly to compute. A recent study [1] 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 [4]—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 [19] 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 1 illustrates how TED can report biased matchings, even on simple trees.

To address these restrictions when attempting to match two web documents, [7] extended TED with some additional move operations executed

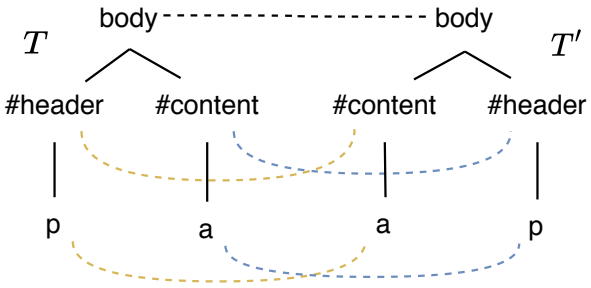

Figure 1: Example of matching biased by the TED (as computed by APTED).

a posteriori to address the ancestry restriction and [14, 13] 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.

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 2 and 3 cover related work, with section 3 focusing in details on the *Flexible Tree Matching* (FTM) original algorithm. Section 4 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 5 thoroughly evaluates our solution against the state of the art on a realistic dataset of web documents. Section 6 discusses the threats to validity of our contribution. Section 7 concludes and overviews some perspectives for this work.

2. 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 [24] 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) [17, 18, 19] 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.

[1] 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* [31, 32] is an edit distance where disjoint subtrees can only be mapped to disjoint subtrees. The *Tree Alignment Distance* [9] is a TED where all insertions must be performed before any deletion. The *Top-Down* distance [21] 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*

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

distance [25] 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. [20] 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 [31]. These restrictions are particularly problematic when matching two full web pages together [13]. 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 [4] 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 [28] favors accuracy over speed and computes an optimal matching using hashings of path signatures. XKeyDiff [6] builds on XyDiff and adds matching logic based on XML primary keys, `XML_SIM_CHANGE` [26] and `XREL_CHANGE_SQL` [23] match XMLs stored in relational databases using SQL. Phoenix [16] 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 [12]. 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 [13], 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.* [13, 14] 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.

3. 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.* [13]. We first describe the notations used throughout the rest of the article, and then describe the main steps of the algorithm.

3.1. 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 $c \prec s$.

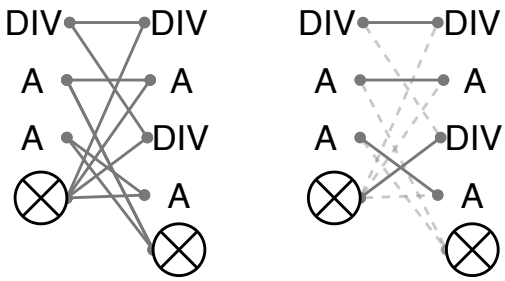

Figure 2: Building a bipartite graph G representing the set of all possible matchings (left) and then compute the optimal full matching (right).

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 2).

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 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

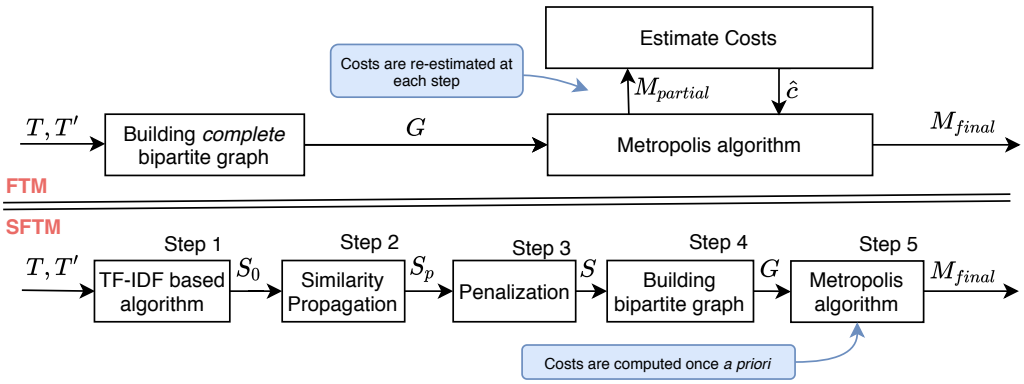

Figure 3: Steps to compute a full matching between two trees T and T'. Upper part covers FTM, while lower part is SFTM.

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 3 describes the main steps involved in computing the final full matching between T and T'.

3.2. 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:

$$c(e) = \begin{cases} w_n & \text{if } n \text{ or } n' \in \{\Theta, \Theta'\} \\ w_r c_r(e) + w_a c_a(e) + w_s c_s(e) & \text{otherwise} \end{cases} \quad (1)$$

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 3). The need to re-estimate the approximated costs after each edge selection actually imposes some critical limitations on the scalability of the algorithm.

3.3. Metropolis *Algorithm*

Finding the optimal matching, given the graph G and the cost function c is a challenging problem, the authors even proved in [14] that this problem is NP-hard. Consequently, the authors described how to use the Metropolis algorithm [15] to approximate the optimal matching. The Metropolis algorithm provides a way to explore a probability distribution by random walking through samples. FTM uses this algorithm 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:

$$f_{FTM}(M) = \exp(-\beta\ c(M)) \tag{2}$$

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 5), notably due to the continuous re-estimation of the approximated costs at each step of the full matching generation.

3.4. Complexity Analysis

The original FTM article [14] 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 4.3).

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.

4. 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 3). 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 5).

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.

4.1. 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 3 illustrates the different steps described in this section.

4.1.1. Initial Similarity (step 1)

To compute the initial similarity s_0 between N and N' (cf. *step 1* in Figure 3), 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 4.2.2 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 [10] and more precisely, the *Inverted Document Frequency* (IDF) part of the formula. Applying TF-IDF to our similarity yields the following definition:

$$IDF(tk) = log(|N|/|TM[tk]|) \tag{3}$$

$$s_0(n, n') = \sum_{tk \in TK} IDF(tk) \tag{4}$$

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 4.2, 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 4.3, we discuss how $f(N)$ influences the worst-case theoretical complexity.

4.1.2. 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 [11].

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:

$$s_i(n, n') \leftarrow s_{i-1}(n, n') + w_i \times s_{i-1}(p(n), p(n')) \tag{5}$$

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

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

$$s_i(p(n), p(m)) \leftarrow s_{i-1}(p(n), p(n')) + v_i \times s_{i-1}(n, n') \tag{6}$$

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 5, leading to high accuracy on a large variety of web documents.

4.1.3. 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':

$$s(n, n') = s_P \times (1 - penalty(n, n')) \tag{7}$$

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

$$penalty(n, n') = \frac{|(|children(n)| - |children(n')|)|}{max(|children(n)|, |children(n')|)} \tag{8}$$

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').

4.1.4. 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 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:

$$c_{SFTM}(e) = \begin{cases} w_n, & \text{if } n \text{ or } n' \text{ is a no-match node} \\ \frac{1}{1+s_p(n,n')}, & \text{otherwise} \end{cases} \tag{9}$$

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.

4.2. 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.

Figure 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.

4.2.1. 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 4 (2a,b) depicts two examples of inverted index. We note $TMmap[key]$ the set of $nodes$ whose key in TM is key. In Section 4.2.2, 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 1 on each node $n' \in N'$. In Algorithm 1, 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 5, we can now easily compute s_p and s.

4.2.2. 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:

$$tokens(n) = \{xPath(n), \texttt{tag}, att_1..a_l, tok(val_1)..tok(val_l)\} \qquad (10)$$

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.

4.2.3. Building G (step 4)

Using Equation 9, 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 $values(neighbours(n')) \subset N$.

4.2.4. 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:

$$f_{SFTM}(M) = \exp(-\beta\ \frac{c(M)}{|M|}) \tag{11}$$

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 2,

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 $edge \in E$).

In practice, we first compute all the connected nodes and edges before storing them as dictionaries, so that the function $connectedEdges$ in Algorithm 2 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 [14]: $\gamma = 0.8, \beta = 2.5$ and a number of iterations of 10.

4.3. 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

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 $\leftarrow remainingEdges.first$
    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

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 $O(N \cdot \sqrt{N} \cdot log(N))$.

5. 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.

5.1. 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 [2, 5, 9, 25, 27, 31] 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.

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 5 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

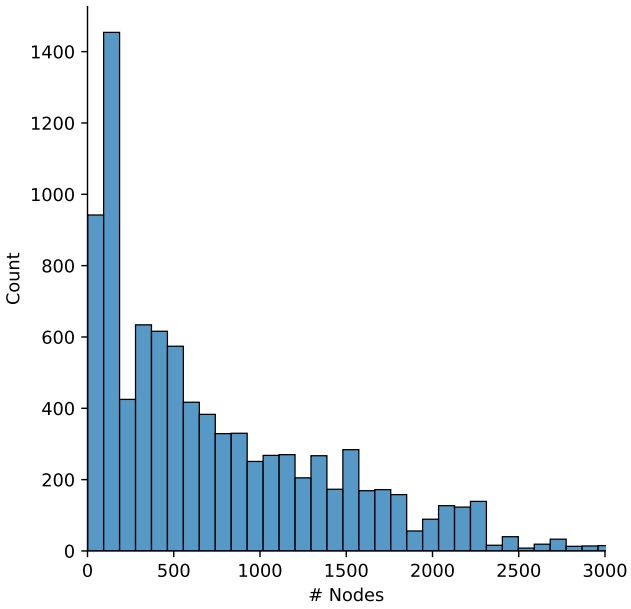

Figure 5: Distribution of DOM sizes (in terms of nodes) in the dataset.

5.2. 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 [19, 18]. Since APTED yields the optimal solution to the TED problem, TED is theoretically superior in accuracy to all more restricted solutions (see Section 2).

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.

5.3. 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 $M_a \subset N \times N'$ 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 6 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

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

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 7 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.

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 [16]—*i.e.*, the number of good matches produced per millisecond. Figure 8 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.

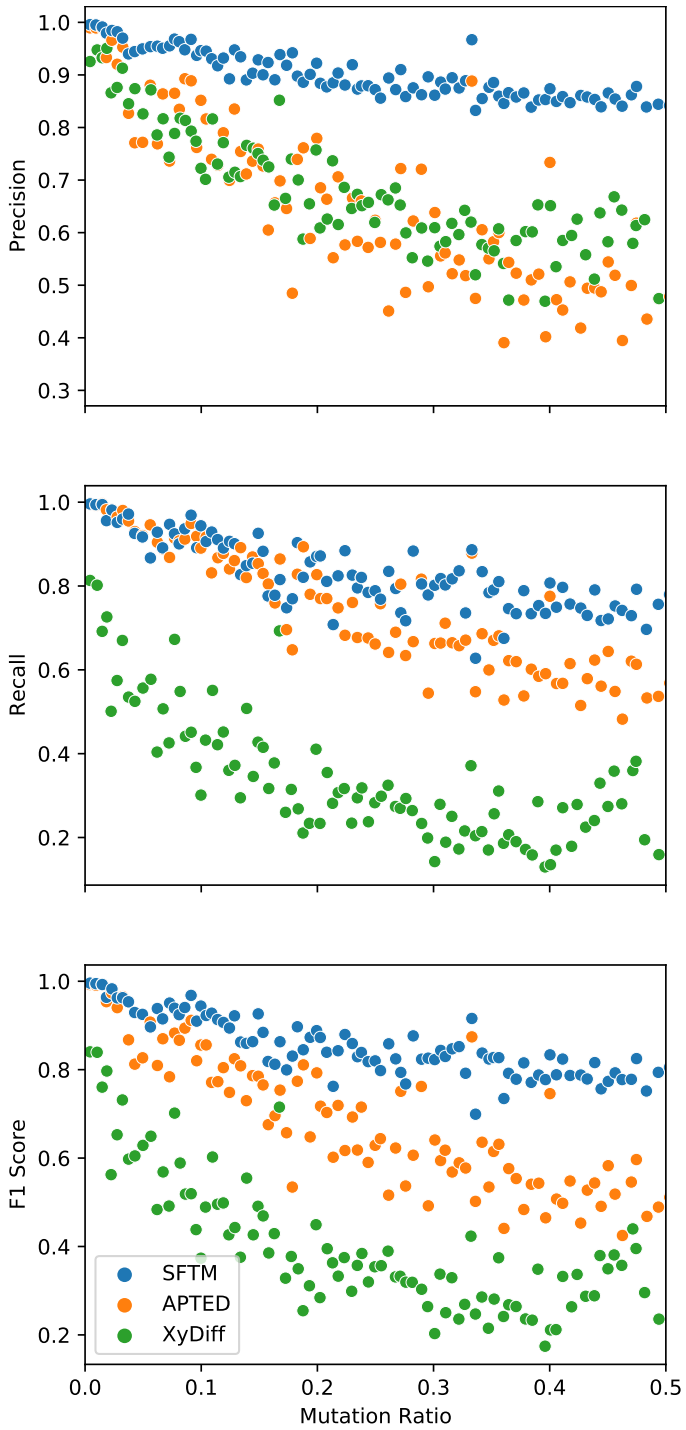

Figure 6: Precision, Recall, and F1 Score of SFTM, APTED, and XyDiff.

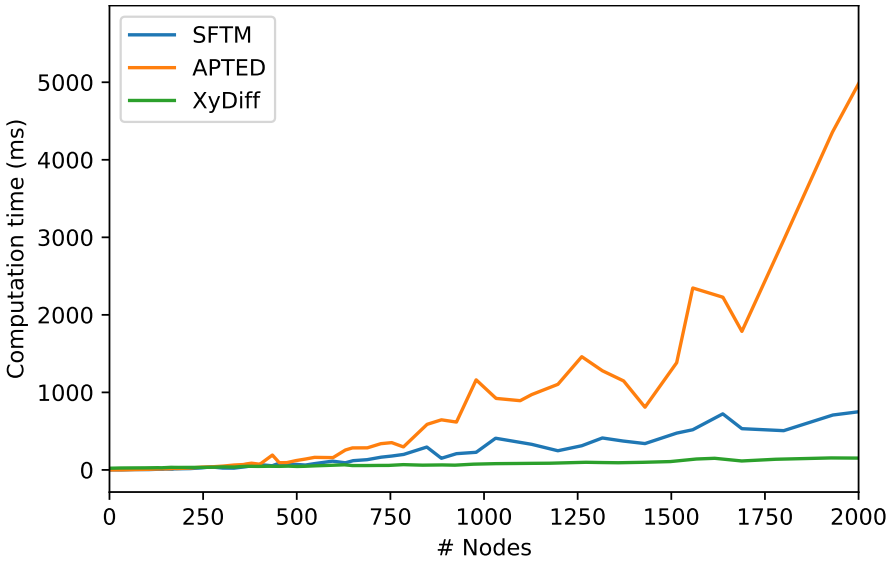

Figure 7: Computation times when matching trees of different sizes.

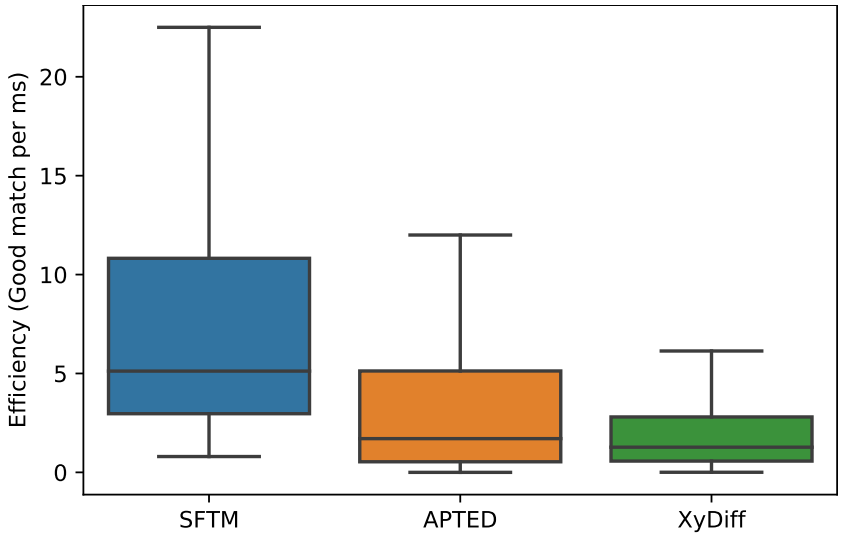

Figure 8: Matching efficiency of SFTM, APTED, and XyDiff.

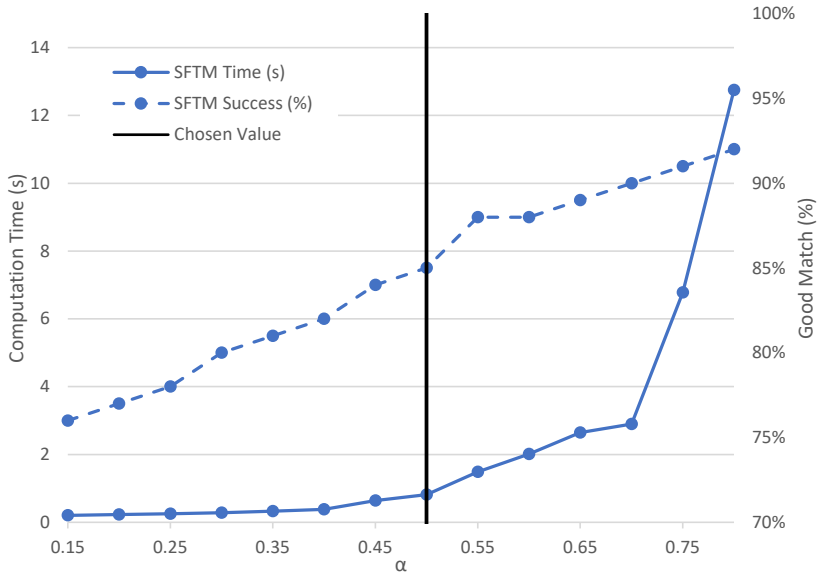

Figure 9: Performance of SFTM given $f(N) = N^{\alpha}$ according to α.

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 9, 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.

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 5.3), 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.

6. 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.

7. 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 expensive. [14] 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.

References

[1] K. Bringmann, P. Gawrychowski, S. Mozes, and O. Weimann. Tree edit distance cannot be computed in strongly subcubic time (unless apsp can). In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1190–1206. Society for Industrial and Applied Mathematics, 2018.

[2] H. Bunke and K. Shearer. A graph distance metric based on the maximal common subgraph. Technical report, 1998.

[3] S. R. Choudhary, D. Zhao, H. Versee, and A. Orso. Water: Web application test repair. In *Proceedings of the First International Workshop on End-to-End Test Script Engineering*, pages 24–29. ACM, 2011.

[4] G. Cobéna, S. Abiteboul, and A. Marian. Detecting changes in XML documents. *Proceedings - International Conference on Data Engineering*, pages 41–52, 2002.

[5] Y. Dinitz, A. Itai, and M. Rodeh. On an Algorithm of Zemlyachenko for Subtree Isomorphism. Technical report, 1998.

[6] R. C. Dos Santos and C. Hara. A semantical change detection algorithm for xml. *SEKE 2007*, page 438, 2007.

[7] M. Fokaefs, R. Mikhaiel, N. Tsantalis, E. Stroulia, and A. Lau. An empirical study on web service evolution. In *2011 IEEE International Conference on Web Services*, pages 49–56. IEEE, 2011.

[8] Y. Hao and Y. Zhang. Web services discovery based on schema matching. In *Proceedings of the thirtieth Australasian conference on Computer science-Volume 62*, pages 107–113. Australian Computer society, Inc., 2007.

[9] T. Jiang, L. Wang, and K. Zhang. Alignment of trees—an alternative to tree edit. In *Annual Symposium on Combinatorial Pattern Matching*, pages 75–86. Springer, 1994.

[10] K. S. Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation*, 1972.

[11] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[12] H. W. Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

[13] R. Kumar, J. O. Talton, S. Ahmad, and S. R. Klemmer. Bricolage: example-based retargeting for web design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2197–2206. ACM, 2011.

[14] R. Kumar, J. O. Talton, S. Ahmad, T. Roughgarden, and S. R. Klemmer. Flexible tree matching. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence (AAAI)*, 2011.

[15] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21(6):1087–1092, 1953.

[16] A. Oliveira, G. Tessarolli, G. Ghiotto, B. Pinto, F. Campello, M. Marques, C. Oliveira, I. Rodrigues, M. Kalinowski, U. Souza, et al. An efficient similarity-based approach for comparing xml documents. *Information Systems*, 78:40–57, 2018.

[17] M. Pawlik and N. Augsten. RTED: a robust algorithm for the tree edit distance. *Proceedings of the VLDB Endowment*, 5(4):334–345, 2011.

[18] M. Pawlik and N. Augsten. Efficient computation of the tree edit distance. *ACM Transactions on Database Systems (TODS)*, 40(1):1–40, 2015.

[19] M. Pawlik and N. Augsten. Tree edit distance: Robust and memory-efficient. *Information Systems*, 56:157–173, 2016.

[20] D. d. C. Reis, P. B. Golgher, A. S. Silva, and A. Laender. Automatic web news extraction using tree edit distance. In *Proceedings of the 13th international conference on World Wide Web*, pages 502–511. ACM, 2004.

[21] S. M. Selkow. The tree-to-tree editing problem. *Information processing letters*, 6(6):184–186, 1977.

[22] A. Stocco, M. Leotta, F. Ricca, and P. Tonella. Apogen: automatic page object generator for web testing. *Software Quality Journal*, 25(3):1007–1039, 2017.

[23] S. Sundaram and S. K. Madria. A change detection system for unordered xml data using a relational model. *Data & Knowledge Engineering*, 72:257–284, 2012.

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

[25] G. Valiente. An efficient bottom-up distance between trees. In *spire*, pages 212–219, 2001.

[26] W. Viyanon and S. K. Madria. Xml-sim-change: structure and content semantic similarity detection among xml document versions. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 1061–1078. Springer, 2010.

[27] J. T. Wang and K. Zhang. Finding similar consensus between trees: An algorithm and a distance hierarchy. *Pattern Recognition*, 34(1):127–137, 1 2001.

[28] Y. Wang, D. J. DeWitt, and J.-Y. Cai. X-diff: An effective change detection algorithm for xml documents. In *Proceedings 19th international conference on data engineering (Cat. No. 03CH37405)*, pages 519–530. IEEE, 2003.

[29] X. Yao, B. Van Durme, C. Callison-Burch, and P. Clark. Answer extraction as sequence tagging with tree edit distance. In *Proceedings of the 2013 conference of the North American chapter of the association for computational linguistics: human language technologies*, pages 858–867, 2013.

[30] Y. Zhai and B. Liu. Web data extraction based on partial tree alignment. In *Proceedings of the 14th international conference on World Wide Web*, pages 76–85. ACM, 2005.

[31] K. Zhang. Algorithms for the constrained editing distance between ordered labeled trees and related problems. *Pattern recognition*, 28(3):463–474, 1995.

[32] K. Zhang. A constrained edit distance between unordered labeled trees. *Algorithmica*, 15(3):205–222, 1996.