Optimal Schemes for Robust Web Extraction

Aditya Parameswaran
Stanford University
adityagp@cs.stanford.edu

Nilesh Dalvi
Yahoo! Research
ndalvi@yahoo-inc.com

Hector Garcia-Molina
Stanford University
hector@cs.stanford.edu

Rajeev Rastogi
Yahoo! Labs Bangalore
rrastogi@yahoo-inc.com

ABSTRACT

In this paper, we consider the problem of constructing wrappers for web information extraction that are robust to changes in websites. We consider two models to study robustness formally: the *adversarial* model, where we look at the worst-case robustness of wrappers, and *probabilistic* model, where we look at the expected robustness of wrappers, as web-pages evolve. Under both models, we present efficient algorithms for constructing the provably most robust wrapper. By evaluating on real websites, we demonstrate that in practice, our algorithms are highly effective in coping up with changes in websites, and reduce the wrapper breakage by up to 500% over existing techniques.

1. INTRODUCTION

Several websites use scripts to generate HTML populated with information from structured backend databases, including shopping sites, entertainment sites, academic repositories, library catalogs, as well as form-based websites. The structural similarity of script-generated webpages makes it possible for information extraction systems to use simple rules to extract information from all the webpages in the website. Such rules are called *wrappers*, and the problem of inducing wrappers from labeled examples has been extensively studied [3, 11, 7, 16, 6, 1, 8]. The information thus extracted may then be used to recreate parts or all of the database.

As an example, we can use wrappers to extract information about restaurants and reviews from multiple aggregator sites like Yelp (`yelp.com`) and Zagat (`zagat.com`). Once a wrapper is learnt for each site, it can be used to obtain a continuous feed of new reviews and restaurants added to the sites, as well as keep up-to-date restaurant information such as hours of operation and phone numbers. As another example, comparison shopping websites use wrappers to continuously obtain a feed of product listings from a large number of merchant websites. Since wrappers can frequently be generated with relatively few labeled examples, wrappers have become a successful and even dominant strategy for extracting information from script-generated pages.

Wrapper Breakage Problem : As wrappers rely heavily on the structure of the webpages to extract data, they suffer from a fundamental problem: the underlying webpages frequently change, and even very slight changes cause the wrapper to break and require them to be re-learned.

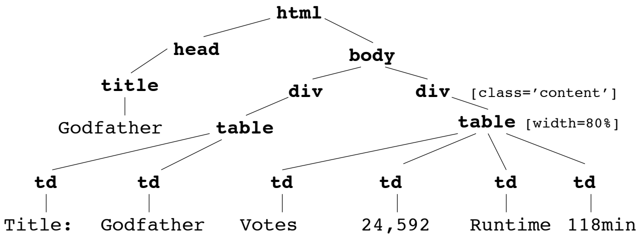

Figure 1: An HTML Webpage

To illustrate, we reproduce an example from [17] in Figure 1, that represents an XML document tree of a script-generated movie page. If we want to extract the number of votes, we can use an XPath such as the following:

$W_1 \equiv$ `/html/body/div[2]/table/td[2]/text()` (1)

However, there are several small changes that can break this wrapper, e.g. if the first `div` is deleted or merged with the second `div`, a new `table` or `tr` is added under the second `div`, the order of `Votes` and `Runtime` is changed, a new font element is added, and so on.

Wrapper breakage has been widely acknowledged as a problem in several empirical studies [8, 2, 14, 15]. From our own experience with a large scale industrial information extraction system at Yahoo! Research [18], we observed that wrappers learnt without robustness considerations had an average life of 2 months, with 1 out of every 50 wrappers breaking every day on average.

Thus, in order to deal with wrapper breakage, there is a need to manually re-label (i.e., provide the locations on the page containing the structured information of interest) the webpages fairly often and to re-learn the wrapper using the newly labeled data. This is a laborious and costly process.

Prior Work on Robustness : Myllymaki and Jackson [8] observed that certain wrappers are more robust than others, and in practice, can have significantly lower breakage. For instance, the following two XPaths can be used as an alternative to W_1 in Eq. (1) to extract the number of votes.

$W_2 \equiv$ `//div[@class='content']/*/td[2]/text()`
$W_3 \equiv$ `//table[@width='80%']/td[2]/text()`

Intuitively, these wrappers exploit more *"local"* information than W_1, and are immune to some of the changes that break W_1. Myllymaki and Jackson constructed robust wrappers manually, and left open the problem of learning such rules automatically. The first formal framework to capture the notion of robustness of wrappers was recently proposed by Dalvi et al. [17]. They define a model

Proceedings of the VLDB Endowment, Vol. 4, No. 11
Copyright 2011 VLDB Endowment 2150-8097/11/08... $ 10.00.

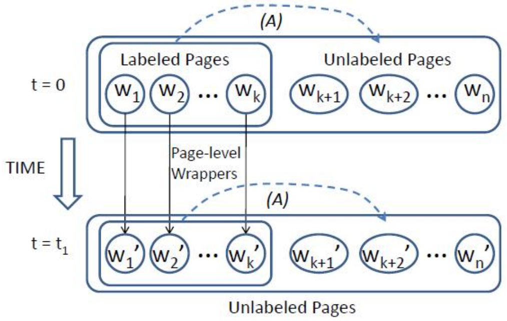

Figure 2: Overview of our approach.

to capture how webpages evolve over time, and the model can be used to evaluate the robustness of wrappers, such as the ones above, in a principled way. However, while their techniques enable us to choose between a set of alternative XPaths by evaluating their robustness, the problem of constructing the most robust wrapper was left open. This problem is the focus of our work.

Our Problem : In this work, we consider the problem of constructing wrappers with *provably optimal robustness*. In constructing a robust wrapper, the choice of the language used to express wrappers plays an important factor. For instance, if we go beyond simple XPath rules, one might start from a set of complementary XPaths that extract the same information, and use majority voting. The resulting wrapper will break only if more than half of the XPaths fail. Thus, with more powerful languages, such as a set of complementary XPaths, we can derive more robust wrappers. In this paper, we pose the following question: *what is the most robust wrapper for a given set of webpages without any constraint on the representation language?*

Our Approach : To illustrate our approach, consider Figure 2, which illustrates the tasks of a wrapper learning system. The website at time $t = 0$ consists of a set of pages $\{w_1, \cdots, w_n\}$, out of which a small subset of pages are labeled (i.e., the locations of the information of interest has been provided by a human annotator), say $\{w_1, \cdots, w_k\}$ for $k < n$. The website at $t = t_1$ consists of the new versions $w'_1, \dots, w'_n$ of $w_1, \dots, w_n$ respectively.

There are two extraction tasks involving wrappers: (A) extraction from the rest of the website $w_{k+1}, \dots, w_n$ by generalizing from the k labeled examples $w_1, \dots, w_k$, and (B) extraction from a future version of the website $w'_1, \dots, w'_n$ by being robust to structural changes. The previously proposed approaches [8, 17] obtain wrappers, written in some specific language, that work for both tasks (A) and (B).

However, the two extraction scenarios (A) and (B) are very different, and impose different robustness and performance requirements on wrappers. In scenario (A), the website contains thousands or millions of pages with similar structure; thus, we need wrappers that are extremely fast and have the ability to withstand small structural variations within pages. Any standard wrapper algorithm [3, 11, 7, 16, 6, 1] can be used to solve task (A).

In contrast, since page structure can change significantly over time, scenario (B) requires highly robust wrappers capable of withstanding much larger structural changes. In order to solve task (B), we proceed in two steps: Given a new version $w'_1, \dots w'_n$ of the website, for each labeled page $w_1, \dots, w_k$ in the old version of the website, we compare the w_i to w'_i to determine the location of the label in w'_i. Subsequently, once we have the labels of $w'_1, \dots, w'_k$, we can train any standard wrapper algorithm (that we used to solve task (A)) to extract from $w'_{k+1}, \dots, w'_n$. Note that only the first step involving label transfer needs to be highly robust. Additionally, we can trade off performance to achieve higher robustness in this step since labels are transferred for only a small number of labeled pages. Furthermore, this label-transfer procedure is relatively space-efficient since we only need to maintain a small number of labeled pages of the old version (i.e., $w_1, \dots, w_k$, instead of $w_1, \dots, w_n$.)

Thus, in order to tackle the robustness issue in task (B), we simply need to focus on the problem of labeling a new version of a webpage given a labeled old version. For this problem, we can look at a webpage w holistically to obtain a robust wrapper specific to the page, rather than relying on generalized site-level features written in some wrapper language. As a result, we are able to design wrappers that achieve a substantially higher robustness. We call this the *page-level robust wrapper* problem.

Page-level Wrapper Robustness : The problem is formally defined in Section 2, and is the focus of this paper. We are given a page w along with the locations of the labeled nodes, and we want to extract the information from a future version of w. We consider a change model that captures how webpages evolve over time by looking at the likelihood of all possible changes. Change models for webpages have been previously proposed [17], along with algorithms for learning models based on archival data. Given a model, we give algorithms to construct a wrapper that gives the most likely locations of the labels in the new version. We consider two different notions of robustness : *probabilistic robustness*, that looks at wrappers that are most likely to work in the future in expectation, and *adversarial robustness*, that looks at wrappers that are most likely to work in the future in the worst-case.

Summary of our contributions :

1. We propose a new approach to wrapper robustness, using *page-level robust wrappers*, enabling us to construct optimally robust wrappers independent of the wrapper language.

2. We analyze our algorithms formally, and show that they are provably optimal under certain models of changes. We define two models, *probabilistic* and *adversarial*. We show that we can construct optimal wrappers under both models efficiently in PTIME. We prove that computing the robustness of these optimal wrappers is NP-Hard.

3. For both models of change, we also provide an estimate of our confidence in the extraction, which can be effectively used to identify candidate websites for retraining of wrappers. This is a novel feature of our approach, as no existing wrappers provide confidence or reliability estimates.

4. We conduct experiments on real websites, and demonstrate that our algorithms are highly effective in practice in coping up with changes in websites. We reduce the breakage rate of wrappers by a factor of up to 5 over traditional wrappers, leading to a significant reduction in the cost of maintaining wrappers. In addition, our confidence estimates provide a very good indication as to when our extraction may be incorrect.

Organization : The paper is organized as follows. In Section 2, we formally define the page-level wrapper robustness problem. We give the solution to the adversarial case in Section 3 and the probabilistic case in Section 4. Our experimental evaluation is presented in Section 5. Additional related work can be found in Appendix E.

2. PROBLEM DEFINITION

Ordered Labeled Trees : Let w be a webpage. We represent w as an ordered, labeled tree corresponding to the parsed HTML DOM tree of the webpage. As an example, consider Figure 1, representing the HTML of an IMDB page. The children of every node are ordered, (e.g., in Figure 1, the node corresponding to `head` is ordered before the node corresponding to `body` among the children of the root) and every node has a label from a set of labels $\mathcal{L}$ (e.g., the root has a label `html`). The label of a node essentially indicates the type of the node.

In addition, the nodes at the leaves are text nodes (depicted in Figure 1 as text in gray). For instance, the first leaf has textual content "`Title`". Since we are primarily interested in structural changes, for our algorithms we replace all HTML text nodes with nodes having special label "TEXT". (In Appendix B.4.1, we describe some extensions that we use to leverage the textual content information in addition to structure.)

We define two webpages w_1 and w_2 to be *isomorphic*, written as $w_1 \equiv w_2$, if they have identical structure and labels, i.e. there is a bijection b between the nodes in w_1 and w_2 that respects labels and order. Two nodes $n_1 \in w_1$ and $n_2 \in w_2$ are said to be isomorphic ($n_1 \equiv n_2$), if $w_1 \equiv w_2$ and n_1 and n_2 are mapped to each other under the bijection.

We assume that each webpage w has a distinguished node $d(w)$ containing the textual information of interest, e.g., the node containing the number of votes on an `imdb.com` movie page. For ease of presentation, we assume that there is a single distinguished node in each page. Appendix C describes how our techniques can be used to handle multiple distinguished nodes, e.g., extracting the list of actors in a movie page.

Edit Operations : We are interested in modeling the scenario when the webpages undergo structural changes. Each change is one of three edit operations: insertion of a node (i.e., insert a node x as a child of another node y, and assign some subsequence of y's children as x's children,) deletion of a node (i.e., delete node x and assign all of x's children as its parent's,) and substitution of the label of a node.

Each edit operation takes an ordered labeled tree and creates a new ordered labeled tree, i.e., the labels and structure of the new tree are the same as in the old tree, except for the single node that was edited (either inserted, deleted, or substituted). Thus, apart from one node that may have been inserted or deleted, there is an implicit *mapping* between the old and new versions of the nodes in the two trees. Furthermore, these mappings can be "composed" across various edits, effectively providing a mapping between the first and last trees for a sequence of edit operations. Note that the only nodes in the first tree that do not have a mapping are those that are deleted at some point in the sequence of edits, and the only nodes in the last tree that do not have a mapping are those that were inserted at some point in the sequence of edits.

A sequence s of edit operations is defined to be an *edit script*. We let $s(w)$ denote the new version of the webpage obtained by applying the operators in s in sequence to w. We use $s(n), n \in w$, to denote the node in $s(w)$ that n maps to when edit script s is applied. (Note that we are overloading function s, however the use should be clear from the context.)

Evolution of Webpages : We assume there is some evolution process π that takes a webpage w and creates a new version of the webpage $\pi(w)$ by performing a sequence of edit operations on w that include insertion of nodes, deletion of nodes and substitution of labels. Thus π is essentially an edit script. Since we are primarily interested in structural changes, we ignore any changes that are not to the ordered labeled tree. However, we are not given what π is; we are only provided the new version of the webpage $w' = \pi(w)$.

Wrappers and Robustness : Let w be a webpage with a distinguished node $d(w)$. We want to construct a wrapper that extracts from future versions of w. Let $w' = \pi(w)$ be a new version of the webpage. We want to find the location of distinguished node in w'.

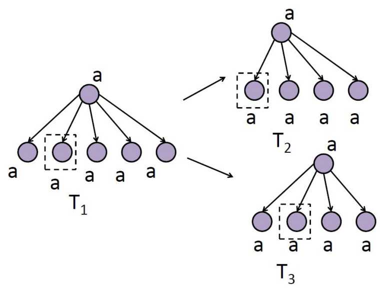

Figure 3: Adversarial vs. Probabilistic Wrappers

We assume that the distinguished nodes are never deleted when webpages evolve. For instance, an IMDB movie webpage will always contain the number of votes even though the webpage may change. It is reasonable to assume this since the content of interest is usually an important part of the webpage, and we only intend to be robust to cosmetic changes and other peripheral changes (e.g., changes to ads) that do not affect the main content. Thus, there is a single distinguished node $d(w') = \pi(d(w))$ in the new tree $w' = \pi(w)$, namely the node that the distinguished node $d(w)$ is mapped to on applying edits π.

As an example, consider Figure 3. Every node in the ordered labeled trees in this figure has the label "a". The second leaf in T_1 is the distinguished node (with a dashed box around it). Consider tree T_1 and T_2. Let us say tree T_2 is obtained from T_1 by an edit script π that deletes the first leaf in T_1. Then, the distinguished node in T_2 is now the first leaf (displayed using a dashed box in T_2.) On the other hand, tree T_3 is obtained from T_1 by an edit script π that deletes one of the last 3 leaves of T_1. The distinguished node in T_3 is now the second node. Note that T_2 and T_3 are isomorphic trees.

We are now ready to define what we mean by a wrapper.

> A *wrapper* is a function ϕ from a webpage to a node in the webpage. We say that a wrapper ϕ *works* on a future version w' of a webpage w, denoted $\phi \models w'$, if $\phi(w') = d(w')$.

If $\phi(w') \neq d(w')$, then we say that the wrapper has *failed* or has *broken*. As an example, consider the wrapper "extract the second leaf" for Figure 3. If we apply this wrapper to T_2, it fails. On the other hand, if we apply it to T_3, it works.

Our objective is to construct page-level wrappers which are immune to π, i.e., wrappers which continue to extract the distinguished node in the new versions of webpages. We consider two different models for the evolution process π.

Adversarial Robustness : In the adversarial model, we assume that each edit operation has an associated cost. (The formal definition for edit costs is given in Section 3.1.)

> We define the *adversarial robustness* of a wrapper ϕ on page w as the largest cost c such that for all w' that can be obtained from w using a set of edit operations with total cost at most c, we have $\phi(w') = d(w')$.

Thus, we want wrappers that are guaranteed to work when we bound the changes to the underlying webpage, i.e., we want our wrapper to work for the "worst-case" modification of cost c. This leads to our first problem:

PROBLEM 1. *Given a page w, compute the wrapper ϕ that has the highest adversarial robustness on w.*

We denote the optimal wrapper by $\phi_{adv}[w]$, or simply ϕ_{adv} when w is clear from the context. Not only do we desire to find ϕ_{adv}, we

Model	Optimal Wrapper (Single Node)	Optimal Wrapper (k- Nodes)	Optimal Robustness	Confidence (Single Node)	Confidence (k- Nodes)
Adversarial	$O(n_1 n_2 d_1 d_2)$	$O(n_1 n_2 d_1 d_2)$	NP-Hard	$O(n_1 n_2 d_1 d_2)$	$O(k n_1 n_2 d_1 d_2)$
Probabilistic	$O(n_1 n_2^2 d_1)$	$O(n_1 n_2 \times \max(n_2 d_1, k d_2))$	NP-Hard	$O(1)$	$O(1)$

Table 1: Complexity of various problems under the two models. Here, n_1 and n_2 are sizes of the trees corresponding to the old and new versions of the webpages, and d_1 and d_2 are the depths.

also want to evaluate its robustness, as it gives a measure of how good any wrapper can be for the given page. In particular, the robustness of the optimal wrapper can be used as an indication as to how often the page needs to be recrawled. If the robustness is low, then the wrapper may break even on small cost changes to the webpage. For such a case, we would want to recrawl the fresh version of the webpage fairly often in order to ensure correctness. Note that once we apply our wrapper on a new version and extract the distinguished node, we can update the wrapper accordingly (assuming the extraction is correct).

PROBLEM 2. *Compute the adversarial robustness of ϕ_{adv}.*

Finally, given a new version w' of webpage w, we want to evaluate the *confidence* in our extraction from w'. Confidence is a measure of how much we trust our extraction of the distinguished node for the given new version. (We define confidence formally in Section 3.5.) For instance, if w' differs significantly from w, we might have less confidence in our extraction. However, if all the differences between w and w' are in a different part of the page, distant from the distinguished node, we might have a larger confidence in our extraction. Our next problem is:

PROBLEM 3. *Given a new version w' of w, compute the confidence of ϕ_{adv} on w'.*

Informally, robustness is a measure of how well wrappers will work on future versions of a given webpage, while confidence is how well a wrapper will work on a particular future version.

Probabilistic Robustness : While the adversarial robustness measures the behavior in the worst case, probabilistic robustness measures the behavior of a wrapper in the expected case.

We illustrate this distinction with the example in Figure 3. Probabilistically speaking, if one node is deleted, it is much more likely that one of the three leaf nodes after the distinguished node in T_1 get deleted, rather than the single node before the distinguished node. In this case, a probabilistically optimal wrapper would return the second leaf instead of the first leaf as the distinguished node (since the second leaf is more likely to be the distinguished node.)

On the other hand, if we were to consider the worst case (adversarial) behavior, then both these trees were derived by a single delete operation from T_1, and therefore both edit scripts have the same edit cost. Thus, an adversarial wrapper would consider the first or the second leaf given a tree with structure T_2 (or equivalently T_3) to have equal chance of being the distinguished node.

In the probabilistic model (defined formally in Section 4), we assume that the edit operations in π come from some probability distribution. In other words, for a webpage w, $\pi(w)$ gives a probability distribution over all possible future states of w.

Given such a probabilistic change model, we define the *probabilistic robustness* of a wrapper ϕ on page w as simply the probability that $\phi \models w'$, where w' is drawn randomly from the distribution $\pi(w)$.

We denote the optimal wrapper by $\phi_{prob}[w]$, or simply ϕ_{prob} when w is clear from the context.

We have the following three corresponding problems for the probabilistic model.

PROBLEM 4. *Given w, compute the optimal wrapper ϕ_{prob}.*

PROBLEM 5. *Compute the probabilistic robustness of ϕ_{prob}.*

PROBLEM 6. *Given a new version w' of w, compute the confidence of ϕ_{prob} on w'.*

We present solutions to these 6 problems in Sections 3 and 4. Table 1 contains the complexity for each of the problems for the two models. The multiple-node wrapper is the version where we want to extract k distinguished nodes from the new version of the webpage. (We consider it in Appendix C.)

Storage of State : Since we operate at a page-level, a wrapper can be constructed by simply storing the old version of the webpage w as well as the distinguished node $d(w)$. Subsequently, on being provided a new version w', the wrapper executes a procedure using three arguments $w, w', d(w)$ to output the node in w' that is the distinguished node. In other words,

$$\phi(w') := f(w, d(w), w') \qquad (2)$$

All our optimal wrappers are of this form. Note that storing the old version of the webpage might incur an additional storage cost — however, the human cost of re-annotation when a wrapper breaks far outweighs the small cost of storage in order to achieve optimality. In addition, we only need to store the ordered labeled tree and not the entire HTML content of the webpage, a small fraction of the size of the webpage.

3. ADVERSARIAL MODEL

In this section, we consider the adversarial robustness of wrappers. We first derive an upper bound on the robustness in Section 3.2. Subsequently, we design a wrapper that achieves the upper bound in Section 3.3. We prove that finding the maximum robustness is intractable in Section 3.4. Then, we describe a measure of confidence for the adversarial model, and discuss how to evaluate it in Section 3.5.

3.1 Preliminaries

Edit Costs : We described the possible edit operations to a webpage in Section 2. We now describe adversarial edit costs for these operations. Let $\mathcal{L}$ be the set of all labels. We assume that there is a cost function e, from $\mathcal{L} \cup \{\emptyset\} \times \mathcal{L} \cup \{\emptyset\} \rightarrow \mathcal{R}$, such that $e(l_1, l_2)$ represents the cost of substitution of a node with label l_1 to one with label l_2, $e(\emptyset, l)$ denotes the cost of insertion of a node with label l, and $e(l, \emptyset)$ represents the cost of deletion of a node with label l. Note that $e(l, l) = 0$. In addition, we assume that the triangle inequality holds, i.e., $e(l_1, l_3) \leq e(l_1, l_2) + e(l_2, l_3)$.

Given an edit script s, the cost of s, denoted $cost(s)$, is simply the sum of costs of each of the operations in s as given by the cost function e.

Adversarial Wrapper : Now we consider the adversarial robustness problem. Given a webpage w with a distinguished node $d(w)$, we want to construct the wrapper ϕ_{adv} that has the maximum adversarial robustness. This wrapper is constructed by storing the webpage w as well as the distinguished node $d(w)$, as discussed in Equation 2 in Section 2. Subsequently, the wrapper is then a function that takes in a new version w' of the webpage w, executes a procedure utilizing w',w and $d(w)$ and returns the distinguished node in the new version.

3.2 Upper Bound

We first derive an obvious upper bound on the adversarial robustness. We define the *ambiguity* of a page, denoted $amb(w)$, to be the smallest number c such that there are two edit scripts s_1 and s_2, with $cost(s_1) \leq c$, $cost(s_2) \leq c$, $s_1(w) \equiv s_2(w)$ but $s_1(d(w)) \not\equiv s_2(d(w))$. In other words, there exists two scripts, each with cost less than or equal to c, such that both the scripts result in isomorphic trees, but map the distinguished node to different nodes in the trees.

It is easy to see that $amb(w)$ is an upper bound on robustness, since a wrapper cannot distinguish between two pages $s_1(w)$ and $s_2(w)$ (since they are isomorphic), and the distinguished nodes in the two pages differ. Thus, we have:

LEMMA 3.1. *The robustness of ϕ_{adv} is at most $amb(w)$.*

3.3 Achieving the Upper-Bound

We now describe a very simple wrapper ϕ that achieves the upper bound, which shows that the maximum adversarial robustness is precisely the ambiguity of the webpage $amb(w)$.

For any new version of the webpage w', the wrapper computes the edit script s with the smallest cost such that $s(w) \equiv w'$ and returns $s(d(w))$, i.e., the node in w' that node $d(w)$ in w maps to on applying s. (We discuss the details of the algorithm after the following theorem.) We call this wrapper algorithm ϕ_{edit}.

THEOREM 3.2. *The adversarial robustness of ϕ_{edit} on w is equal to $amb(w)$.*

PROOF. By Lemma 3.1, we know that the robustness of ϕ_{edit} is $\leq amb(w)$. In what follows, we show that robustness of ϕ_{edit} is $\geq amb(w)$, and hence the robustness must be exactly $amb(w)$.

Assume on the contrary that the wrapper fails on a webpage w', where w' was obtained from w using a script s_{actual} with $cost(s_{actual}) < amb(w)$. Thus, we have $w' = s_{actual}(w)$. Since the wrapper fails on w', $\phi_{edit}(w') \not\equiv s_{actual}(d(w))$.

Let s_{min} be the smallest cost edit script that takes w to w'. Thus, we have $s_{min}(w) \equiv w' \equiv s_{actual}(w)$. Also, by definition, $s_{min}(d(w)) \equiv \phi_{edit}(w')$. Since s_{min} has the smallest cost, we have $cost(s_{min}) \leq cost(s_{actual}) < amb(w)$. Also, $s_{min}(d(w)) \equiv \phi_{edit}(w') \not\equiv s_{actual}(d(w))$, so $s_{min} \not\equiv s_{actual}$. Thus, we have two different scripts, s_{min} and s_{actual}, both with cost less than $amb(w)$, such that they result in isomorphic trees but different distinguished nodes. This contradicts the definition of ambiguity. □

COROLLARY 3.3. *The ϕ_{edit} wrapper achieves the maximum adversarial robustness, which is equal to $amb(w)$.*

We can use an algorithm by Zhang and Sasha [10] to compute the minimum cost tree edit script from w to w'. Note that the original algorithm was only presented for edit distances which satisfy *metric* properties, i.e., the edit costs satisfy the triangle inequality and the cost of deletion of a node with a given label is same as the cost of insertion of a node with the same label. However, the algorithm can be shown to also work under weaker conditions, with the only constraint being the triangle inequality between edit costs.

The complexity of the algorithm is $O(n_1 n_2 d_1 d_2)$, where n_1 and n_2 are the number of nodes in the trees w and w', while d_1 and d_2 are the maximum depths of the nodes in w and w'.

3.4 Computing Maximum Robustness

Section 3.3 settles Problem 1. Now we look at Problem 2, computing the maximum adversarial robustness. We observe a surprising result that although we can construct the wrapper with maximum adversarial robustness in PTIME, computing its robustness is NP-hard. We obtain a reduction from the NP-Complete partition problem. The proof may be found in Appendix A.1.

THEOREM 3.4. *Computing $amb(w)$ is NP-hard.*

3.5 Confidence in Extraction

Now we consider Problem 3, determining the confidence in our extraction given a new version w'. Note that confidence we define is not one in the traditional sense. In particular, it is not a normalized number in $[0, 1]$, but is simply a non-negative number. However, it does give an indication as to how good the extraction is. If the confidence is large, the extraction is likely to be correct.

Intuitively, if the page w' differs a lot from w, then our confidence in the extraction should be low. However, if all the changes in w' are in a distinct portion away from the distinguished node, then the confidence should be high despite those changes. Based on this intuition, we define the *confidence* of extraction on a given new version as follows.

Let s_1 be the smallest cost edit script that takes w to w' (i.e., $s_1(w) \equiv w'$). Thus, the node extracted by ϕ_{edit} is $s_1(d(w))$. We also look at the the smallest cost script s_2 that takes w to w' (i.e., $s_1(w) \equiv s_2(w) \equiv w'$) but does not map $d(w)$ to the node corresponding to $s_1(d(w))$. We define the confidence as the difference $cost(s_2) - cost(s_1)$. In other words, this quantity is nothing but the additional cost that needs to be used to break the optimal adversarial wrapper. Intuitively, if this difference is large, the extracted node is well separated from the rest of the nodes, and the extraction is unlikely to be wrong.

For instance, consider the example in Figure 3. In the figure, if we used the wrapper "extract the second leaf of T_2", then the confidence of the wrapper is 0, since there is an equal cost edit script (i.e., the one on top), that gives a different distinguished node.

To compute the confidence, we need to solve the following *constrained edit script* problem : given two trees w and w' and two nodes n and n', compute the smallest edit script between w and w' with the constraint that n is not mapped to n'. It is straightforward to extend the Zhang-Sasha algorithm [10] for solving the unconstrained minimum cost edit script problem to solve the constrained version. Computing the confidence has the same complexity as computing the pairwise tree edit cost i.e., $O(n_1 n_2 d_1 d_2)$.

The complete algorithm for extraction and confidence computation (Algorithm 1) can be found in the appendix.

3.6 Discussion

In this section, we provided the optimal wrapper under adversarial model with a given set of edit costs. This brings us to the issue of setting the edit costs. Intuitively, the edit cost for an operation should capture how difficult it is for a website maintainer to make that edit operation. For instance, inserting a new table `table` should be more expensive than inserting a new row `tr`. There are techniques [17] that learn the frequencies with various edit operations take place in websites, which can be used to derive the costs. For instance, we can set the cost of an edit to be (1 - probability of that edit). In the experiments, we show that even the simple model that assigns a unit cost to each edit operation performs exceedingly well in practice.

In the next section, we use probabilities of change to create an optimal wrapper that works best in an expected case.

4. PROBABILISTIC ROBUSTNESS

We now analyze the robustness of wrappers under a probabilistic model. In Section 4.1, we define what we mean by a probabilistic edit model. We derive a specification of the optimal probabilistic wrapper in Section 4.2. The design of an efficient wrapper according to the specification is done in Section 4.3. (Some of the technical details may be found in Appendix B.) Similar to the adversarial case, deriving robustness of the optimal wrapper is shown to be intractable in Section 4.4. We also show how to derive a confidence estimate for our extraction in the same section.

4.1 Probabilistic model

We consider an existing probabilistic model for tree-edits [17], which was proposed recently to capture how webpages evolve over time. In this model, probabilities are defined in terms of a *probabilistic transducer* π, which takes as input a webpage w, and outputs a new version w' of the webpage by performing a random set of edit operations drawn from a probability space. These edit operations are the same as the ones described in Section 2.

As before, let $\mathcal{L}$ be the set of all labels. The transducer is parametrized by a set of probabilities : the deletion probability $p_{del}(l)$ for each label $l \in \mathcal{L}$, the insertion probability $p_{ins}(l)$ for each label l, and the substitution probability $p_{sub}(l_1, l_2)$ for each pairs of labels l_1, l_2. In addition, it takes a probability p_{stop} (indicating the probability that the transducer stops making edits to the webpage). Additional details about the operation of the transducer can be found in Appendix B.1. The resulting transducer defines a probability distribution over all possible future states of the webpage w, which we denote by $\pi(w)$.

Given two trees w_1 and w_2, with sizes n_1, n_2 and depths d_1 and d_2, Dalvi et al. [17] provide an algorithm that computes the probability $\mathbf{P}(\pi(w_1) \equiv w_2)$ in time $O(n_1 n_2 d_1 d_2)$. They also provide an efficient algorithm to learn the parameters of the probabilistic transducer from archival data. Here, we assume that such a probability model is already given to us.

4.2 Optimal Probabilistic Robustness

We now consider Problem 4 which is as follows : given a probabilistic transducer π and a webpage w, construct the wrapper ϕ_{prob} that maximizes the probability $\mathbf{P}(\phi_{prob} \models w')$, where w' is drawn from the probability distribution $\pi(w)$. In this section, we provide the specification of the wrapper and a simple but inefficient algorithm to meet the specification. In the next section, we design a more efficient algorithm.

Given two webpages w_1 and w_2, along with two nodes m_1 and m_2 in respective webpages, define $\mathbf{P}_\pi(w_1, w_2, m_1, m_2)$ to be the probability that the transducer π transforms w_1 to a tree isomorphic to w_2 with the constraint that node m_1 gets mapped to a node isomorphic to m_2. For the purposes of the section, we use "a transforms into b" instead of "a transforms into a tree isomorphic to b" for simplicity. Similarly, we say "m_1 gets mapped to m_2", instead of "m_1 gets mappped to a node isomorphic to m_2".

The specification of the wrapper is similar to that of the optimal algorithm π_{adv} for the adversarial case. Define ϕ_{prob} to be the wrapper that given a new version w' of a webpage w, outputs the node x in w' that maximizes $\mathbf{P}_\pi(w, w', d(w), x)$. In other words, we wish to compute:

$$\arg\max_{x \in w'} \mathbf{P}_\pi(w, w', d(w), x) \tag{3}$$

The following result is straightforward.

THEOREM 4.1. *ϕ_{prob} is the wrapper with highest probabilistic robustness.*

Observe that for a fixed x, we can reduce the problem of computing $\mathbf{P}_\pi(w, w', d(w), x)$ to the standard problem of computing $\mathbf{P}(\pi(w) \equiv w')$ for a pair of trees. We can simply change the labels of $d(w)$ and x to new unique labels l_1^* and l_2^* and impose a probability of 0 for deletion of label l_1^* and for all substitutions of l_1^* except to l_2^*. Thus, we can compute $\mathbf{P}_\pi(w, w', d(w), x)$ in time $O(n_1 n_2 d_1 d_2)$. By letting x vary over all possible nodes in w_2, repeating this procedure immediately gives us an algorithm for an optimal probabilistic wrapper with complexity $O(n_1 n_2^2 d_1 d_2)$.

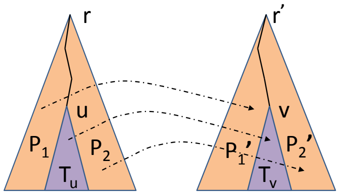

Figure 4: Probabilistic Wrapper: Mappings between sub-trees

4.3 Achieving Optimal Robustness

In the naive algorithm described in the previous section, different calls to $\mathbf{P}_\pi(w, w', d(w), x)$ in Eq. 3 share lots of computations for various x, so we can eliminate the redundancy to make the algorithm more efficient and reduce its complexity. We describe this algorithm in this section.

Given a node in a tree, we define its *prefix* to be all nodes in the tree that are to its left or to the left of its ancestors. In other words, if we were exploring the nodes in the tree in a post-order fashion, (i.e., children of a node before the node itself) the nodes that are visited before the given node is visited form its prefix. Note that P_1 is a prefix of the node u in the tree in the left in Figure 4. Any prefix of a node in a tree is also regarded as a prefix of the tree itself.

Now, let w' be a tree obtained from w using a sequence of edit operations. Consider the canonical mapping between the nodes of w' and w, which keeps track of where the nodes in w ended up in w'. We formally defined this mapping in Section 2. Our basic observation is that this mapping respects sibling and parent relationships, as well as the prefix order. For instance, if node a maps to b, then a child of a can only map to a child of b. Similarly, a node to the left of a can only map to a node to the left of b. For instance, in Figure 4 which depicts the two trees, if node u gets mapped to node v, then the nodes in P_1 must be mapped to nodes in P_1', nodes in T_u must be mapped to nodes in T_v, and nodes in P_2 must be mapped to nodes in P_2'.

We have the following result:

THEOREM 4.2. *For any $u \in w, v \in w'$, $\mathbf{P}_\pi(w, w', u, v)$ equals the product of: (according to the notations of Figure 4.)*

- p_{1v} *: the probability that P_1 transforms to P_1'.*
- p_{2v} *: the probability that T_u transforms to T_v.*
- p_{3v} *: the probability that P_2 transforms to P_2'.*

Note that in Figure 4, the ancestors of u and v are contained in P_2 and P_2', and as a result, the probabilities P_1 and P_2 are not symmetric (i.e., P_2 is not a prefix of u if the ordering of the nodes were reversed in the tree.)

The basic idea behind the more efficient algorithm is to precompute the transformation probabilities between all pairs of prefixes of complete sub-trees, and to use this to compute $\mathbf{P}_\pi(w, w', d(w), v)$ for all possible v. The analysis is non-trivial, and for lack of space, we give the details of the algorithm in Appendix B. We state here the complexity of the algorithm.

THEOREM 4.3. *The probabilistic wrapper ϕ_{prob} has complexity $O(n_1 n_2^2 d_1)$.*

Thus, we get an $O(d_2)$ speed-up over the naive algorithm. In practice, this makes a big difference, and makes extraction from some of the large websites tractable. In Appendix B.4, we describe some heuristics and optimizations that speed up the algorithm even further, and make it very efficient.

4.4 Evaluating Robustness and Confidence

Robustness: Let ϕ_{prob} denote the optimal wrapper as outlined in the previous section. We define the robustness of this wrapper on a given webpage as the probability that, given a page w, ϕ_{prob} works on w', where w' is drawn from a distribution given by π. Similar to the adversarial case, we can show the following result, once again using a reduction from the partition problem (We omit the proof.)

THEOREM 4.4. *Computing the robustness of the probabilistic wrapper ϕ_{prob} is* NP*-hard.*

Confidence: Also, given a new page w', we want to compute the confidence in our extraction from w'. We define confidence of extraction to simply be the probability of a correct extraction, given that w' is derived from w using a probabilistic edit script π.

The probability that the extraction is correct is exactly the probability $\mathbf{P}_\pi(w, w', d(w), \phi_{prob}(w'))$. Since the algorithm for the probabilistic wrapper also computes this quantity, we get it for free.

5. EVALUATION

In this section we evaluate our wrappers against each other and against traditional XPath-based techniques.

5.1 Experimental Setup

Data: To test the robustness of our techniques, we used archival data from Internet Archive (`archive.org`), a website storing previous versions of webpages on the Internet going back several years. We obtained webpages from Internet Archive for three different domains:

- A: IMDB Movie Database (`imdb.com`)
- B: CNN Stock Price Listings (`money.cnn.com`)
- C: Wikipedia (`en.wikipedia.org`)

(We have also experimented with other domains: Yahoo! Finance (`finance.yahoo.com`), and Citysearch (`citysearch.com`), and the results are not very different from the ones presented for the three domains above.) For each of these websites, for our experiments, we chose a set of ($\approx$ 10) webpages that have archival versions. For each of these webpages, we crawled every version found on Internet Archive (typically of the order of several 100's of versions per webpage.) Additional details on the data sets including crawl times, number of pages, number of versions, and so on can be found in Appendix D.1.

Distinguished Nodes: In order to test wrappers, we need to detemine if an extraction is "correct." One approach is to manually inspect each extraction result to check if the wrapper found, say, the correct book price or movie title. This approach does not scale well, so we developed the following alternative scheme.

In each of our data sets, we manually select distinguished nodes that can be identified because they either have a fixed unique textual content or have a portion of their content that is fixed and unique across all versions. For instance, in the IMDB pages, the number of user votes for a movie is always displayed as "<Number> votes". As an other example, in the CNN Stock Price Listings, the distinguished node containing the last traded stock price always contains the string "last price". Similarly, in Wikipedia, the time zone of a city always contains the content "time zone".

In our experiments, we implicitly *hide* this fixed textual content from the wrappers, but not from our evaluation code. After the wrapper produces its result, we can then easily check if it is correct by seeing if the result contains this fixed textual content.

In a way, we are artificially transforming easy extraction problems into to hard extraction problems. That is, if this textual content is visible, then even a simple wrapper can simply look for this textual content on the webpage and return the corresponding node. But without this content, the wrapper must rely on less clear indications in the webpages. We believe that our synthetic test cases are representative of the more challenging extraction cases where wrappers tend to fail, and have the added advantage that it is easy to check for correctness.

Implementation: We implemented both the adversarial (denoted ADV in the figures) and probabilistic wrappers (denoted PROB in the figures) in Java. For ADV, we used unit costs for each edit operation. In additional experiments, not shown here, we observed that the results are not very sensitive to the precise cost choices made — for instance, the results for setting the cost of changes to be (1 - their probabilities) are very similar. For PROB, we used the techniques described previously [17] to learn the probabilities of every edit operation on a dataset of IMDB pages. We then use these probabilities as input to the wrapper.

Baselines: For comparison, we implemented two other wrappers. The first uses the full XPath containing the complete sequence of labels of the nodes from the root to the distinguished node (in the initial version of the webpage). This wrapper is often used in practice. The second uses the robust XPath wrappers from [17]. We will call these wrappers FULL and XPATH respectively.

Objectives: The input to a wrapper is (a) the old version of a page, (b) the location of the distinguished node in that page, and (c) the new version. For each execution, we check if the wrapper finds the distinguished node in the new version. We also study the usefulness of the confidence values returned by the wrappers.

We study how well our wrappers perform as a function of the elapsed time between the old and the new versions of the webpage. We use *skip* sizes as a proxy for this time. We say a pair of versions has a *skip* of k if the difference between the version numbers of the two versions is k.

We also study the relative performance between PROB and ADV for all possible skip sizes, and the average edit distance, time taken, and size of trees as a function of the skip size. Since these experiments are not central to the paper, they can be found in Appendix D.

In addition, we also measure the time taken for extraction by each of our wrappers. Since our wrappers run offline, time is not critical; however, our wrappers run reasonably fast. The wrapper PROB takes 500ms on average on a processor running Intel Core 2 Duo 1.6GHz, while ADV takes 40ms on average. (See Appendix D for more details.)

Evaluation of Robustness : In this experiment, we vary the skip size, and we evaluate all wrappers for how many mistakes are made in extraction. In particular, we study the total number of errors for three skip sizes ($k = 1, 5$ and 10) for each of the techniques. For each skip size k, for all pairs of version numbers $(i, i + k)$ (for all version numbers i), we ran all the wrappers, and plotted the results in Fig 5(a). (For instance, if there were 10 versions, for $k = 5$ our pairs of versions are $(1, 6), (2, 7), \ldots (5, 10)$.) The y axis represents the percentage of times the wrapper failed (in log scale). For instance for $k = 1$, the full XPath wrapper and the robust XPath wrapper failed around 15% and 5% of the time while the PROB and ADV both perform similarly and failed in less than 1% of the cases. Thus, our wrappers perform much better.

Figures 5(b) and 5(c) contain the results for the *Wikipedia* and *CNN* datasets respectively. The results for Wikipedia and CNN are similar, with ADV and PROB outperforming all other methods. For instance, on the Wikipedia data, for $k = 10$, PROB has one in four

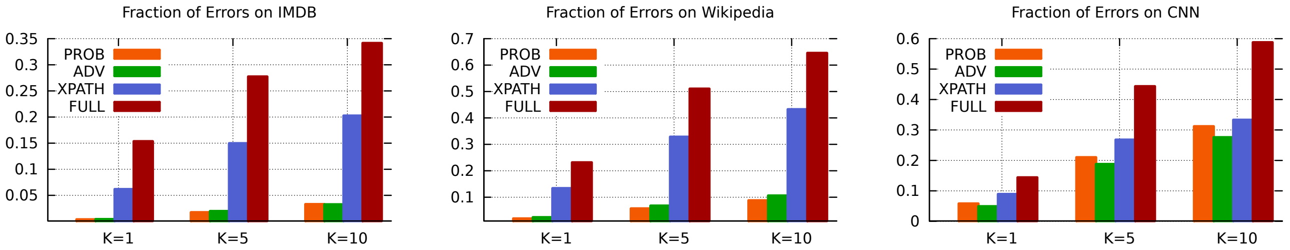

Figure 5: Experiments on (a) IMDB (b) Wikipedia (c) CNN

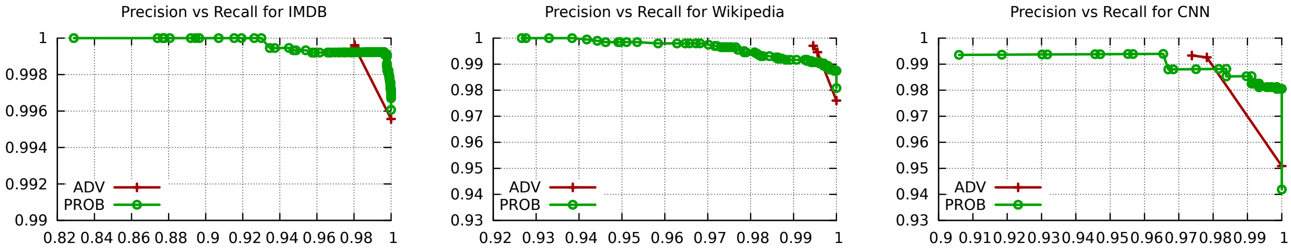

Figure 6: Precision-Recall curves for Adv and Prob for (a) IMDB (b) Wikipedia (c) CNN

errors that XPath has. Between Adv and Prob, there is no clear distinction. Both are comparable to each other, with Prob being better on the Wikipedia and IMDB webpages, and Adv having a slight edge on the CNN webpages. On the CNN pages, all wrappers perform slightly poorly, possibly because of high variability in the structure of the website. (However, the confidence estimates of our wrappers (as we see next) are fairly reliable, even for CNN.)

Accuracy of the Confidence Threshold: In addition to being robust, our wrappers also output confidences on the extractions. The confidence values are an indication of when the wrappers are making mistakes. These values can be used to decide whether to use the extracted results, or to refer the webpage for human relabeling.

In this experiment, we study the accuracy of the confidence values in predicting the correctness of the extractions for Adv and Prob. We define our "relevant" set to be precisely the instances where the extraction is correct, independent of threshold. We now try to see how accurately these "correct" instances can be retrieved by setting a given confidence threshold.

We evaluate accuracy as follows: Suppose we run the optimal wrapper on a set of pages N. Out of these, let the set N_c be the ones where extraction is correct. In addition, we get a confidence value on each of the pages. We want to use the confidence scores to identify the set N_c of correct extractions. Given a threshold τ on the confidence, let N_τ be the set of pages where the confidence of extraction was higher than τ. Then, we define the precision to be $|N_c \cap N_\tau|/N_\tau$, i.e., the fraction of retrieved instances that were correct, and recall to be $|N_c \cap N_\tau|/N_c$, i.e., the fraction of the total correct instances that were retrieved.

We plotted precision (on the y-axis) vs. recall (on the x-axis) for Adv and Prob in Figure 6. As can be seen from the figures, for both Adv and Prob, the curve is very close to the ideal precision-recall curve, i.e. the confidences are very indicative of correct extraction results. Also, while Prob does not completely dominate Adv, it behaves much better. So even though both Prob and Adv make similar number of mistakes in Figure 5, the confidences in the probabilistic wrapper are much more predictive of the mistakes.

6. CONCLUSION

In this paper, we presented algorithms to construct provably optimal wrappers under the *adversarial* and *probabilistic model*. By evaluating on real websites, we demonstrated that our wrappers are highly effective in coping with changes in websites, and reduce the wrapper breakage by up to 500% over existing techniques, and provide reliable near-perfect confidence estimates.

We showed that both the wrappers have comparable perfomance in terms of wrapper breakage. However, the probabilistic wrapper is more effective in predicting its failures.

7. REFERENCES

[1] A. Sahuguet et. al. Building light-weight wrappers for legacy web data-sources using w4f. In *VLDB*, pages 738–741, 1999.

[2] Mari Abe and Masahiro Hori. Robust pointing by xpath language: Authoring support and empirical evaluation. In *SAINT*, 2003.

[3] Tobias Anton. Xpath-wrapper induction by generating tree traversal patterns. In *LWA*, pages 126–133, 2005.

[4] Boris Chidlovskii et. al. Documentum eci self-repairing wrappers: performance analysis. In *SIGMOD*, pages 708–717, 2006.

[5] D. C. Reis et. al. Automatic web news extraction using tree edit distance. In *WWW*, pages 502–511, 2004.

[6] Wei Han, David Buttler, and Calton Pu. Wrapping web data into XML. *SIGMOD Record*, 30(3):33–38, 2001.

[7] Chun-Nan Hsu and Ming-Tzung Dung. Generating finite-state transducers for semi-structured data extraction from the web. *Information Systems*, 23(8):521–538, 1998.

[8] J. Myllymaki et. al. Robust web data extraction with xml path expressions. Technical report, IBM Research Report, May 2002.

[9] K. Lerman et. al. Wrapper maintenance: A machine learning approach. *JAIR*, 18:149–181, 2003.

[10] K. Zhang et. al. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput.*, 18(6), 1989.

[11] N. Kushmerick, D. Weld, and R. B. Doorenbos. Wrapper induction for information extraction. In *IJCAI*, pages 729–737, 1997.

[12] Nicholas Kushmerick. Wrapper verification. *World Wide Web*, 3:79–94, March 2000.

[13] Bing Liu. *Web Data Mining: Exploring Hyperlinks, Contents, and Usage Data*. Springer-Verlag New York, Inc., 2006.

[14] M. Kowalkiewicz et. al. Robust web content extraction. In *WWW*, pages 887–888, 2006.

[15] Marek Kowalkiewicz et. al. Myportal: Robust extraction and aggregation of web content. In *VLDB*, pages 1219–1222, 2006.

[16] I. Muslea, S. Minton, and C. Knoblock. Stalker: Learning extraction rules for semistructured, web-based information sources. In *AAAI: Workshop on AI and Information Integration*, 1998.

[17] Nilesh N. Dalvi et. al. Robust web extraction: an approach based on a probabilistic tree-edit model. In *SIGMOD*, pages 335–348, 2009.

[18] P. Gulhane et. al. Web-scale information extraction with vertex. In *ICDE*, pages 1209–1220, 2011.

[19] Robert McCann et. al. Mapping maintenance for data integration systems. In *VLDB*, pages 1018–1029, 2005.

[20] Valter Crescenzi et. al. Roadrunner: Towards automatic data extraction from large web sites. In *VLDB*, pages 109–118, 2001.

APPENDIX

A. ADVERSARIAL WRAPPER DETAILS

```
Data: $w, w', u$
Result: New Location of $u$ in $w'$ and confidence of extraction
Find minimum cost script $s$ such that $s(w) \equiv w'$;
$v := s(u)$;
$c_1 := c(s)$;
Find minimum cost script $s'$ such that $s'(w) \equiv w'$ and $v \not\equiv s'(u)$;
$c_2 := c(s')$;
Return $v$, $c_2 - c_1$
```

Algorithm 1: Optimal Adversarial Wrapper

A.1 Proof of Theorem 3.4

PROOF. We show a reduction from the partition problem which is known to be NP-complete. In the partition problem, we are given a set of positive integers $x_1, x_2, \ldots, x_n$, with total sum $2S$, and we want to check if this set can be partitioned into two sets with sum S each.

Consider a tree w that has a root node with n children having labels $a_0, a_1, \cdots, a_n$. The distinguished node is a_1. Consider the following edit costs: all the insertion costs are infinity. All deletion costs are infinity, except a_0 and a_n, which have a deletion cost of 0. Additionally, we have a set of labels $b_0, \cdots, b_{n+1}$ such that $e(a_i, b_i) = 0$, $e(a_i, b_{i+1}) = e(a_i, b_{i-1}) = x_i$ for $1 \le i \le n$, and all other substitution costs are infinity. These edit costs satisfy the triangle inequality. The root cannot be deleted or substituted.

There are only two kinds of edit scripts can lead to ambiguity. (These two edit scripts are depicted in Figure 7 leading to T_4.) These are (1) delete a_0, and transform each a_i to either b_{i-1} or b_i for $1 \le i \le n$, and (2) delete a_n and transform each a_i to either b_i or b_{i+1} for $0 \le i \le n-1$. Note that if one of the a_i continue to stay in the final version, it is easy to ascertain where the distinguished node is. (If a_i is present in the final version, the node $i-1$ nodes to the left is the distinguished node.) Additionally, for (1), if a_0 is deleted and then some a_i gets substituted to b_{i+1}, we can once again ascertain where the distinguished node is. (If the ith location has b_{i+2} for some i, one can be sure that the first node is the distinguished node.) A similar reasoning holds for why substitution to b_{i-1} does not happen for any a_i in (2).

Both the scripts lead to a tree w' containing children $c_1, c_2, \cdots, c_n$, with each $c_i = b_{i-1}$ or b_i. In one script, the distinguished node is in the first position, and in another script, it is in second position. Any tree w' of the above form can be achieved using two ambiguous edit script.

We prove that that the partition problem has a solution iff $amb(w) = S$. Consider any new tree w' of the above form resulting from scripts s_1 and s_2, where s_1 deletes the first node and s_2 deletes the last node. If $c_i = b_i$, then it means that s_1 did not pay a price, while s_2 had a cost x_i. If $c_i = b_{i-1}$, then the edit script s_1 had a cost of x_i, while s_2 did not pay a price. Thus, the total cost of the two scripts is $cost(s_1) + cost(s_2) = 2S$, the total sum of all x_i. Hence, $cost(s_1), cost(s_2) \le S$ iff $cost(s_1) = cost(s_2) = S$, and there is a partition of the set of x_i into two sets with sum S each. □

B. PROBABILISTIC WRAPPER DETAILS

B.1 Background: Probabilistic Transducer

The transducer π makes one pass over the entire webpage w, and at each position, probabilistically decides to either insert a node, delete a node, or substitute its label. Formally, it has the following operations:

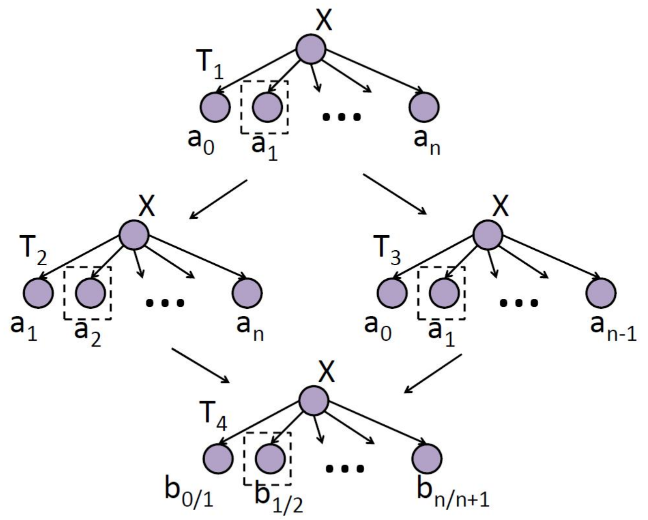

Figure 7: Hardness Proof. First edit script: T_1 followed by T_2 followed by T_4. Second edit script: T_1 followed by T_3 followed by T_4.

1. Let r be the root node of w with label l. First, π is applied recursively to all the children of r to obtain mutated children of r.
2. With probability $p_{del}(l)$, it deletes the node r. Otherwise, with probability $p_{del}(l, l')$, it changes the label of l to l'.
3. With probability p_{stop}, it stops and outputs the tree. Otherwise, with probability $p_{ins}(l_{new})$, it inserts a new node at the top with label l_{new}, assigns a random subsequence of top nodes as the children of the new node, and repeats Step 3.

B.2 Background: TP Algorithm

Since our optimizations try to make use of shared computation in the algorithm provided in [17] to compute $\mathbf{P}(\pi(w) \equiv w')$, we first describe the details of this algorithm (which we call the TP Algorithm (Transformation Probability Algorithm). This algorithm is primarily based on the Zhang-Sasha algorithm [10] for computing tree edit distance.

The algorithm computes, bottom-up, for every pair of nodes $u \in w$ and $v \in w'$, the probability $\mathbf{P}(\pi(w_u) \equiv w'_v)$, where w_u and w'_v are the subtrees rooted at u and v respectively. In Figure 4, this probability is the probability of transformation of T_u to T_v. Additionally, for a given pair of nodes, the algorithm uses dynamic programming to compute the probability that any prefix of the tree under the first node transforms to any prefix of the tree under the second node. For instance, for the trees under r and r' in Figure 4, P_1 and P'_1 are both prefixes (of r and r' respectively), and as a result, during computation of the transformation probability of r and r', the algorithm also computes the probability of P_1 transforming into P'_1.

For any prefix pairs P_1 and P'_1, the transformation probability $\mathbf{P}(\pi(P_1) \equiv P'_1)$ can be expressed in terms of the transformation probabilities of smaller prefixes as well as transformation probabilities of entire sub-trees. The algorithm uses dynamic programming to efficiently compute the transformation probability of all sub-tree pairs. It can be shown that there are $n_1 d_1$ prefixes of w and $n_2 d_2$ prefixes of w', and hence the total complexity of the algorithm is $O(n_1 n_2 d_1 d_2)$, where n_1 and n_2 are the number of nodes in the two trees and d_1 and d_2 are the depths of the two trees.

B.3 An Improved Algorithm

Now we present our improved algorithm for the problem of computing the location of the distinguished node given a probability model of transformation π. As discussed in Section 4, this is the same as:

$$\arg\max_{v \in w'} \mathbf{P}_{\pi}(w, w', u, v),$$

where $u = d(w)$, which does better than the naive algorithm that invokes the TP algorithm n_2 times for each node v in w'. The pseudocode is provided in Algorithm 2.

The algorithm uses Theorem 4.2. First, consider a vanilla run of the TP algorithm. Since this run computes the transformation probabilities for each pairs of prefixes of nodes in the two trees, we can compute the probabilities p_{1v} and p_{2v} in Theorem 4.2 for all v using the single vanilla run. (Note that p_{1v} is nothing but the probability of transformation of the prefix of u relative to w into the prefix of v relative to w'. Also note that p_{2v} is nothing but the transformation probability for the sub-tree under u to the subtree under v.) The first run is depicted in line labeled 1 in Algorithm 2.

The only difficulty is computing p_{3v} for all v. As we noted earlier, p_{3v} and p_{1v} computations are not symmetric: we need a separate technique to compute p_{3v} for all v. (This is because we now need to also account for the ancestors of u and v.)

We define $P_2(u)$ to be this tree for u, and $P_2'(v)$ to be this tree for v. To compute the transformation probabilities between $P_2(u)$ and $P_2'(v)$, we can run a TP algorithm on these two trees. However, this computation still shares a lot of computation with the original TP run.

THEOREM B.1. *[Complete Subtree] Given nodes x and y in $P_2(u)$ and $P_2'(v)$ respectively, the subtrees rooted at x and y are identical to the corresponding subtrees for the original trees, except when x is an ancestor of u and y is an ancestor of v.*

We omit the proof, but instead give the intuition based on Fig 4; the nodes in P_2 and P_2' are precisely the nodes in the right of u and v or their ancestors. The nodes in the right of u and v do not cut the path from the root r and r' to u and v, and thus do not have the trees under them affected. On the other hand, the ancestors of u and v have the left hand portion of their sub-tree cut by the path from the root to u or v.

Thus, in order to compute p_{3v}, i.e. the probability of $P_2(u)$ transforming to $P_2'(v)$, we only need to compute the additional transformation probabilities for the following trees : (1) an ancestor of u to any node in $P_2'(v)$ that is not v's ancestor, (2) any node in $P_2(u)$ that is not u's ancestor to an ancestor of v and (3) two ancestors of u and v. If we do this naively, this approach would give rise to a overall complexity that is the same as the naive algorithm given earlier.

For (1), using Theorem B.1, the subtree under the node in $P_2'(v)$ is the same as the one in the original tree. Since u is fixed, we can simply run the TP algorithm with arguments $P_2(u)$ and the complete tree w' to derive all these transformation probabilities for all v. This needs to be done precisely once. (This run is depicted in line labeled 2 in Algorithm 2.)

For (2) and (3), observe that we are computing the transformation probability between (a) a sub-tree or a prefix of $P_2(u)$ and (b) a prefix of $P_2'(v)$. This can be computed by computing bottom-up, the transformation probabilities between the prefixes of each of the sub-trees in P_2 and the prefixes of $P_2'(v)$, using dynamic programming. (This run for each v is depicted in line labeled 3 in Algorithm 2.)

Since there are $n_1 d_1$ prefixes of subtrees of $P_2(u)$, and n_2 prefixes of $P_2'(v)$, the total time taken for a fixed P_2' is $n_1 d_1 n_2$. Finally, as we vary P_2' for over all possible n_2 nodes, the total complexity is $O(n_2^2 n_1 d_1)$. This leads to an $O(d_2)$ speed-up over the naive algorithm, which in practice, is significant.

The complete algorithm is outlined in Algorithm 2.

```
Data: $w, w', u$
Result: New location of $u$ in $w'$ and confidence of extraction
1 Run $\mathsf{TP}(w, w')$ and record $p_{1v}$ and $p_{2v}$ for all $v$;
  $T_u :=$ tree under $u$;
  $P_1 :=$ prefix of $u$ in $w$;
  $P_2 := w - T_u - P_1$;
2 Run $\mathsf{TP}(P_2, w')$ and record transformation probabilities of $P_2$ to
  complete subtrees in $w'$;
  foreach $v \in w'$ do
      $T_v :=$ tree under $v$;
      $P_1' :=$ prefix of $v$ in $w'$;
      $P_2' := w' - T_v - P_1'$;
      foreach $z \in P_2$ in PostOrder do
3         Compute transformation probabilities of prefix of $z$ in tree
          $P_2$ to tree $P_2'$;
      end
      $p_{3v} :=$ Transformation probability of $P_2$ to $P_2'$;
      $Pr(v) := p_{1v} \times p_{2v} \times p_{3v}$;
  end
  Return node $v$ with largest $Pr$, and $Pr(v)$;
```

Algorithm 2: Optimal Probabilistic Wrapper

B.4 Heuristics and Optimizations

B.4.1 Alignments

We now describe a heuristic technique we use to improve the performance of our probabilistic wrapper as well as the adversarial wrapper. Often, in the old and new versions of the web-page, there are some text strings that appear precisely once in each of the pages. One simple heuristic for both the probabilistic and adversarial wrappers is to first map these unique text strings to each other in two versions of the web-page (i.e., we assume that the node containing the text in the new version was derived from the node containing the text in the old version.) Once we perform this mapping, both versions of the web-page can be partitioned into various sub-trees. (Note that there is a one-to-one mapping between the sub-trees or partitions in the two versions.) Subsequently, if we are interested in a single distinguished node, we can simply run our probabilistic and adversarial wrappers for the partition that the distinguished node resides in (with the corresponding partition in the new version.)

As an example, consider Figure 4. Let the distinguished node in the first tree (corresponding to the original version of the web-page) be in the sub-tree T_u under u, and let the second tree be the new version of the web-page. In that case, if the text strings contained at node u and node v were identical, and they are not found anywhere else in either of the two trees, then we can reduce the problem into finding the distinguished node given two partitions (sub-trees) T_u and T_v.

In our experiments, we found that the size of the original web-page was as large as 4000 nodes, but after using the technique given above, the size of the sub-tree under consideration was reduced to around 100. This gave rise to a large 50X speedup in the time taken for both the wrappers.

B.4.2 Subforest Optimization

In the probabilistic wrapper, the deletion probability of portions of the tree (of the old version) and insertion probability of portions of the tree (of the new version) is used repeatedly in different runs of transformation probability calculation. These portions are precisely prefixes of complete sub-trees of some node (because at any point in the wrapper algorithm, only sub-trees of this form are under consideration). Thus, we do this computation once and store it for lookup later on. We perform this computation for every pair of nodes from the tree in post-order (i.e., the two nodes, and all nodes in between in post-order). While this does not change the complexity of the wrapper algorithm, it gives a speedup of about 3X in practice.

B.4.3 Early Pruning

In the algorithm for the probabilistic wrapper, we can precompute the product $X(v) = p_{1v} \times p_{2v}$ for all v. This computation is relatively efficient. Subsequently, we can retain the largest $Pr(v')$ computed so far, and avoid running step (3) for all v if $X(v) < Pr(v')$, since all those nodes v can never be the one with the largest Pr.

C. MULTIPLE DISTINGUISHED NODES

In this section, we explore the case of k multiple distinguished nodes in the original tree w, all of whose new locations we need to discover in w'. For instance, for a book on Amazon (`amazon.com`), we might want to extract the title, author, price on the page corresponding to the book.

Adversarial Wrapper: In this case, all we need to do is to discover the lowest cost edit script, and use it to discover mappings for every single node in the set of k distinguished nodes. Thus, the complexity is the same as the single target node case.

Similarly, we can compute the confidence associated with each node by computing the edit cost once, and computing, for each of the k-nodes, the cost of the second edit script that maps the distinguished node to a different node. Thus, the complexity is $O(k \times n_1 n_2 d_1 d_2)$.

Probabilistic Wrapper: In the probabilistic scheme, we need to compute the transformation probabilities for each node in the set of target nodes. However, as in the optimal algorithm in Section 4.3, we can share computation for several of the phases.

In Algorithm 2, the run corresponding to Line 1 stays unchanged since it operates on the original versions of the two trees. The run corresponding to Line 2 now needs to compute transformation probabilities for every distinguished node, giving rise to a complexity of $O(kn_1 n_2 d_2)$. For the run corresponding to Line 3, on the other hand, finding transformation probabilities corresponding to every node v still has the same complexity of $O(n_1 n_2^2 d_1)$. However, for the root, the transformation probability will vary based on which distinguished node is under consideration. Thus, we have an additional complexity of $O(n_1 n_2 k)$ since we do this k times. Thus, the complexity is $O(n_1 n_2 \max(kd_2, n_2 d_1))$.

There is no additional complexity if we wish to return the confidence in answers, since that is anyway used to decide which node to return. The complexity of each algorithm is summarized in Table 1.

D. ADDITIONAL EXPERIMENTS AND EXPERIMENTAL DETAILS

D.1 Dataset Details

Additional details on the dataset are available in Table 2. The table displays, for each website we experimented with, the number of webpages in that website that we considered, the average number of versions of each webpage found on Internet Archive, the number of distinguished nodes in those webpages, and the earliest and latest crawl times of the versions found on Internet Archive across all webpages for that website.

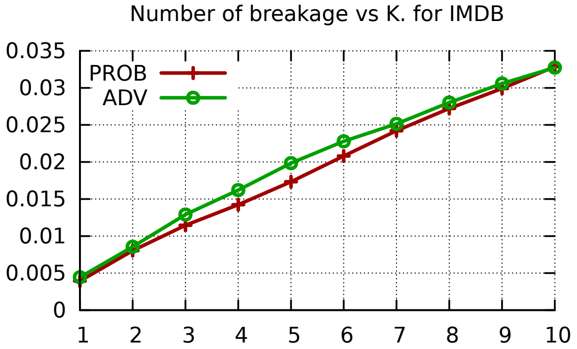

Figure 8: Number of Errors for Probabilistic and Adversarial wrappers

For instance, for IMDB, there were 9 distinct movie pages that we studied over time. On average, each movie page had around 310 archival versions. We studied the performance of our algorithms for 3 different distinguished nodes (corresponding to number of votes, sound format, and number of minutes), the archival versions ranged from October 2003 to June 2008.

Notice that for Wikipedia, the distinguished nodes varied from webpage to webpage, since the Wikipedia webpages were sampled randomly from different topics (ranging from San Francisco to Alexander Graham Bell).

D.2 Relative Performance

We now study closely the performance of the probabilistic wrapper versus the adversarial wrapper as the skip size changes. The results are in Figure 8, where we plot the absolute number of errors for both schemes versus the skip size k for the IMDB dataset. (The IMDB case is especially instructive since the probability model was learnt on a dataset of IMDB pages.) As expected, both schemes perform worse as skip size is increased. Also, as can be seen from the figure, the adversarial and probabilistic wrappers make approximately the same number of mistakes when k is close to 1 or 10, but not around $k \approx 5$, when the difference is approximately 0.5%. The reason for this behavior may be that in the "very easy" (those with $k \approx 1$) and "very hard" cases (those with $k \approx 10$), the changes are either so minimal or so drastic that both the algorithms either catch it or don't. However, for the middle cases, the probabilistic wrapper, since it uses more fine-grained probabilities, does better.

D.3 Variation of Time Taken, Tree Size and Edit Distance with Skip Size

For each of our experiments studying robustness on varying skip size (see Section 5) we also recorded the average size of the trees (after partitioning the trees using the alignment optimization in Appendix B.4.1), as well as the average edit distance between the two archival versions of the trees. This information, along with the average time taken for the probabilistic wrapper is displayed for each data set in Table 3. Note that we do not record the average time taken for the adversarial wrapper because it is always $< 100ms$.

As can be seen in the table, as k increases, so does the time taken, the edit distance as well as the size of the tree. Note that the size of the trees increase as the skip size increases since the alignment optimization of Appendix B.4.1 is able to prune only a small portion of the DOM trees. As a result, the time taken for the

Site	Number of pages	Average number of versions per page	Number of distinguished nodes per page	Earliest crawl time	Latest crawl time
IMDB	9	310	3	02 Oct. 2003	26 Jun. 2008
Wikipedia	11	145	1-3 (Depends on webpage)	04 Dec. 2003	01 Aug. 2008
CNN	11	45	4	08 Feb. 2005	01 Aug 2008

Table 2: Dataset Details (Crawled from Internet Archive)

Parameter	IMDB			Wikipedia			CNN		
	$k = 1$	$k = 5$	$k = 10$	$k = 1$	$k = 5$	$k = 10$	$k = 1$	$k = 5$	$k = 10$
Average time taken (s)	0.44	0.49	0.55	0.44	1.04	1.48	0.45	1.28	1.73
Average edit distance between two versions	1.5	3.7	6	6.5	26.8	38.4	28.7	110.9	130.3
Average size of trees	93.7	97.2	100.4	47.9	60.8	70.2	45.9	94.1	112.7

Table 3: Variation with Skip Size

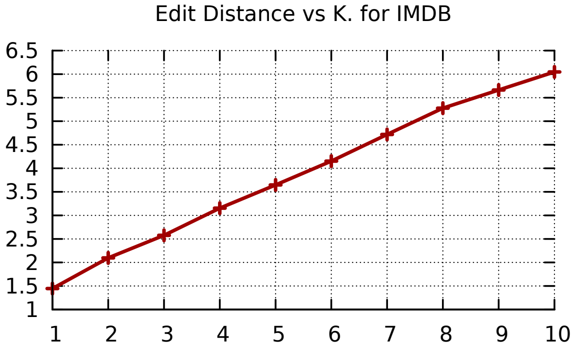

Figure 9: Edit distance vs. Skip Size

probabilistic wrapper increases as well, from 0.45s all the way to 1.73s in some cases. However, even for the hardest case $k = 10$, the average time taken does not go beyond 2s for any website. On the other hand, edit distance predictably increases with step size, since it is a measure of how different the two archival versions of the same webpage are.

To see how much the skip size affects the edit distance, we plotted the average edit distance between two pages versus the skip size for the pages in the IMDB dataset, displayed in Fig 9. This distance is for the trees already partitioned according to the alignment technique in Appendix B.4.1. Note that the edit distance varies almost linearly with respect to the skip size for this dataset. Note also that this edit distance is a proxy for the absolute number of changes around the distinguished node (changes away from the distinguished node will be ignored if we use the alignment heuristic.) This result implies that there are no “sudden changes” happening to IMDB web-pages on average. The changes are gradual, and as long as we make sure to “refresh” our wrapper by extracting the new version periodically, and updating the location of the distinguished node, we should be able to keep extracting correct information.

E. EXTENDED RELATED WORK

Information extraction via wrappers has been studied for more than a decade, with most of the emphasis being on learning wrappers that are *generalizable* or *adaptable* and *correct* [16, 20]. Bing Liu [13] provides a good survey of the area, focusing on web wrappers. However, there has been a recent surge of work on detection and correction of wrappers, on the face of page evolution.

This recent work can be classified into one of three categories (a) wrapper breakage (b) wrapper repair (c) robust wrapper discovery.

The field of *wrapper breakage detection* or *wrapper verification* [12, 19] tries to identify when a wrapper has failed. Typically, verification is done by studying the distribution of content as well as structural features in the two web-pages, and outputting an error whenever the distribution changes drastically. Our algorithms, since they also provide a confidence in the extraction result, can be also used for verification. (If the confidence is too low, we can conservatively output an error and refer to a human annotator for correction.)

Given a wrapper framework that has broken, *wrapper repair* or *wrapper reinduction* [9, 4] tries to automatically re-learn broken wrappers. In particular, Lerman et. al. [9] propose an algorithm to compute statistically significant patterns for attribute values. Changes in these patterns for the extracted values signal that the wrapper is no longer valid. Subsequently, these patterns are once again searched to find the attribute values while repairing broken wrappers. The work on ECI wrappers [4] uses inherent content features (such as word count) to repair wrappers.

Thus, typically, the techniques used in wrapper repair are learning of content models on the web-page, and using it to detect where the new location of the target node is. These content models need to be learned per web-site (since they may not easily generalize from other web-sites), and in some cases, it may not be possible to do so, especially when there is lack of sufficient training data. Our techniques, on the other hand, do not require training data for each website. We simply use an cost model across all sites. However, note that our algorithms (which rely primarily on structure) can only be improved using content models, thus the work on wrapper repair is complementary to ours.

Lastly, there has been some work recently on discovering robust wrappers. However, most of this work either uses human help to help them discover robust wrappers [8], or discovers them from a fixed wrapper language [17].

Mapping one tree to another, or finding the smallest tree edit distance has been used to solve other information extraction problems in the past. It has been used to map similar portions between two trees to each other in order to identify repeating elements in HTML pages [13], as well as identify clusters of web-pages with similar HTML structure [5].