Ranking XPaths for extracting search result records

Dolf Trieschnigg, Kien Tjin-Kam-Jet and Djoerd Hiemstra
University of Twente
Enschede, The Netherlands
{trieschn,tjinkamj,hiemstra}@cs.utwente.nl

ABSTRACT

Extracting search result records (SRRs) from webpages is useful for building an aggregated search engine which combines search results from a variety of search engines. Most automatic approaches to search result extraction are not portable: the complete process has to be rerun on a new search result page. In this paper we describe an algorithm to automatically determine XPath expressions to extract SRRs from webpages. Based on a single search result page, an XPath expression is determined which can be reused to extract SRRs from pages based on the same template. The algorithm is evaluated on a six datasets, including two new datasets containing a variety of web, image, video, shopping and news search results. The evaluation shows that for 85% of the tested search result pages, a useful XPath is determined. The algorithm is implemented as a browser plugin and as a standalone application which are available as open source software.

Categories and Subject Descriptors

H.2.8 [**Database applications**]: Data mining; I.5.4 [**Pattern Recognition**]: Applications—*Text processing*; H.3.5 [**Information Storage and Retrieval**]: Online Information Services—*Web-based services*

General Terms

Algorithms, Experimentation, Performance

Keywords

Web extraction, Scraper, Wrapper, Search result extraction

1. INTRODUCTION

Reusing search results from existing search engines can be useful for a variety of applications. For instance, for building a meta-search engine which aggregates results from multiple sources [28]. Or for automatic analysis and processing of the search results, such as clustering. Reusing search results requires them to be available in a machine readable format. Such a format can be obtained in a number of ways. First, some search engines (such as Google) provide a custom API which allows programmers to directly use search functionality from their program. Second, some search engines provide their search results through (extended) syndication formats, such as RSS and Atom feeds. The OpenSearch standard[1] specifies such extended syndication formats and additionally allows to specify the interface of the search engine. A drawback of both APIs and syndication formats is that these results do not always correspond to the results obtained from the web interface used by ordinary users [30]. Third, the search results can be *scraped* or *extracted* from the webpage presented to the user of a search engine. The search results are typically presented with a combination of HTML, Javascript and Cascading Style Sheets (few search engines provide their results in Flash). An advantage of scraping is that the scraped search results correspond to the results obtained during a normal search session. The actual conversion from the presentation format to a machine readable format is carried out by a *wrapper* (or *scraper*). A search result wrapper for a particular search engine can be 1) manually programmed; 2) constructed interactively with a user; 3) learned from one or more manually labelled search result pages; or 4) fully automatically constructed without any interaction with the user.

The focus of this paper is on automatically constructing a search engine wrapper based on only a single HTML search result page. In contrast to recent work, which proposes fully automatic methods which have to be rerun on each search result page (see Section 2), our method tries to automatically determine a wrapper which can be reused on all result pages based on the same template. The wrapper is specified in terms of a single XPath (XML Path language[2]) expression[3].

Figure 1 illustrates the approach. The figure shows a search result page from Bing web search. In the bottom-left of the figure four possible XPaths to extract search result records are shown. The first, `//li[./div/a]`, specifies 10 nodes from the HTML document highlighted and numbered in the figure. By specifying the nodes in terms of structure and attributes values of its adjacent nodes, the XPath is likely to be reusable for other search result pages from Bing,

[1] http://www.opensearch.org/
[2] http://www.w3.org/TR/xpath/
[3] In the remainder of this paper, we will use the term XPath to refer to an XPath *expression*.

CTIT Technical report
Copyright 2012.

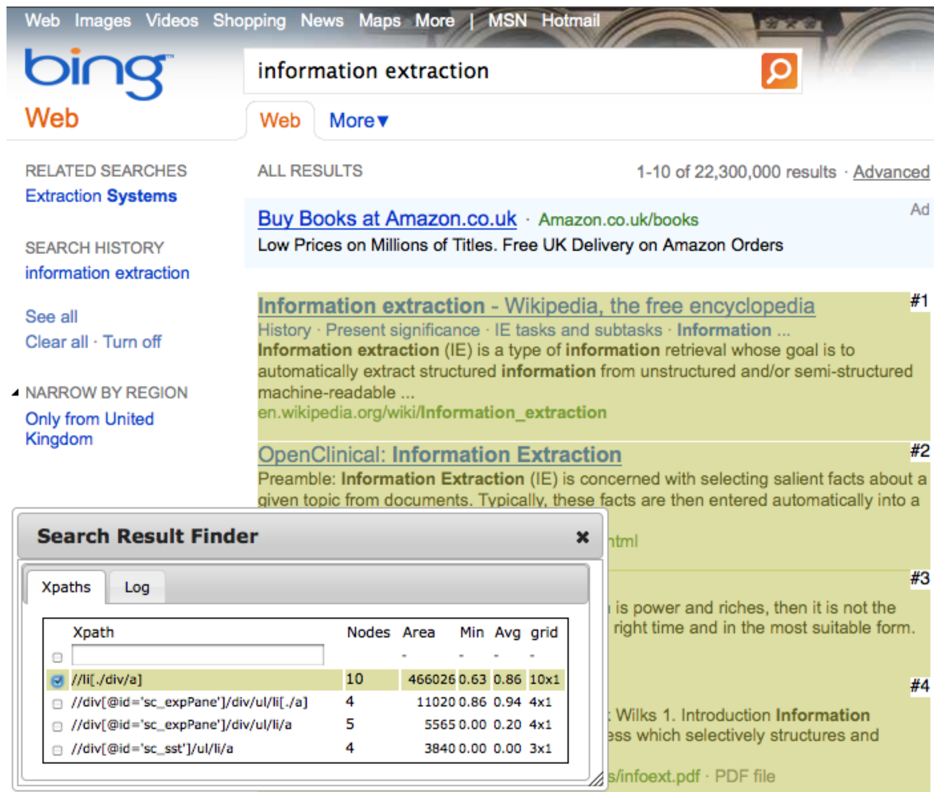


Figure 1: Finding and ranking XPaths for extracting search result records.

including pages which do not include ads. A large advantage of this approach is that the found XPath is portable to other applications: most popular programming languages support executing XPath statements on a Document Object Model (DOM)[4] parsed from an HTML page.

The use of XPaths for web extraction has been previously explored by Myllymaki and Jackson [32]. They emphasize the flexibility of XPaths for extraction and advocate the use of "content-based" (based on text on the webpage), "attribute-based" (the value of node attributes) and "structure-baed" (local node structure) XPaths rather than full paths such as `/html[1]/body[1]/table[3]/tr`. At a more abstract level, these content, attribute and structure-based XPaths provide a partial solution to the "access path dependence" noted by Codd as one of the problems for the integration of information systems [11]. The XPaths depend less on the complete data structure of the HTML page. Myllymaki and Jackson's work is limited to an analysis of *manually* defined XPaths for extraction information from websites. In this work, we *automatically* determine structure-based and attribute-based XPaths.

The contributions of this paper are the following.

- We describe a method to fully automatically obtain reusable search result wrappers based on only a single search result page.
- We evaluate this method on a number of test tests, including a new and up-to-date test set covering a large variety of search results.
- We make the algorithm and test sets publicly available for follow-up research.

[4] `http://www.w3.org/DOM/`

The overview of this paper is as follows. In section 2 related work is presented. In section 3 we describe our approach of determining XPaths for search result extraction. In section 4 the experimental setup is described. In section 5 the results of the evaluation are discussed. We wrap-up with a discussion and conclusion in sections 6 and 7, respectively.

2. RELATED WORK

Before describing related work, we define the terminology we use throughout this paper. We define a *wrapper* as a method to extract records from a webpage. A *record* is some (structured) information about a single entity or object. In case of a *search result record* (SRR), a record represents a single search result. A record consists of multiple *attributes*, pieces of information with a designated meaning. A SRR can consist, for example, of a title, a description and a url attribute. A *template* defines how the records are displayed on a webpage.

Related work can be found in the area of web information extraction. In contrast to conventional information extraction systems, which rely more on natural language processing techniques such as natural language parsing (e.g. [36]), web information extraction leverages additional features available in web documents, for instance the node structure of the webpage.

The term *wrapper* has been used for a large variety of systems, offering different levels of functionality: some wrap-

pers only detect a record (e.g. [17, 43]), some also determine the attributes in each record and align them to each other (attribute alignment, see e.g. [10, 16, 29, 35]) and the last also assign meaningful labels to the attributes (attribute labeling, e.g. [40, 45, 47]). Another distinction can be made in the type of pages the wrapper targets: some aim at extracting information from 'single record' or detail pages (e.g. [4, 38]); many others focus on pages with multiple records.

Early work has shown a focus on *manual* wrapper construction, in which users with varying levels of expertise are involved in manually constructing extraction patterns and scripts (e.g. [5, 12, 20, 34]). Later the development focused on techniques requiring less or less intensive interaction with the user. Based on a number of labelled examples, the systems *induced* a wrapper for pages based on the same template (e.g. [10, 19, 25]). In *interactive* methods, users are continuously involved in the wrapper induction process and users can correct mistakes made by the method (e.g. [6, 45, 46]). The last group of wrappers is fully automatic and can be run without any user interaction (e.g. [2, 9, 27, 35, 40, 42, 43]).

Wrappers also vary in the amount of data they require for operation. Some require multiple example pages (e.g. [14, 44]) to induce a wrapper. The systems relying on manually labelled data vary in the amount of *training* data they require. This varies from a single record example on a single page (e.g. [10, 46]), to records on multiple training pages (e.g. [13, 14, 22, 26]).

Different page features are used to extract records and attributes. Many early wrappers treat the webpage as a sequence of tags and text (e.g. [10, 18]). Later wrappers treat the page as a tree of tags (e.g. [7, 46]). More recently introduced wrappers also involve information about the rendering of parts of the page(e.g. [29, 31, 35, 43, 47]). Also features separate from the page to be wrapped can be used for information extraction. For instance, labels found in search forms associated with the page can be used for attribute labeling (e.g. [37, 40]).

Another distinction between information extraction systems is their output. Some methods *result* in a wrapper (a program) which can be reused to extract records (e.g. [10]). Other methods *are* wrappers: they only return the records they found on the current page (e.g. [21, 29]).

A large variety of techniques is used in extracting records, varying from grammars (e.g. [12]) and grammar learning (e.g. [1, 10]), using patricia trees (e.g. [9]), to the use of similarity measures (e.g. [21]), sometimes in combination with clustering techniques (e.g. [31, 33]). Frequently, heuristics are employed to reduce the algorithmic complexity or simply to improve accuracy (e.g. [17, 21]).

For more extensive surveys on web information extraction systems, see [8, 23, 24].

In this work we build a system for 1) record detection; which 2) is fully automatic; 3) requires only a single page as input; 4) uses the document tree and rendering information as features; 5) outputs a reusable wrapper and 6) is based on similarity measures and heuristics.

A number of researchers have recently investigated the usage of XPaths for information extraction. Anton proposes a method which automatically determines a relative XPath for information extraction based on a set of annotated example documents [3]. In contrast to our method, it allows the construction of complex XPaths, which also include predicates about node siblings. Dalvi et al. propose a method to learn robust XPath wrappers based on noisy example data [16]. Urbansky et al. use XPaths to extract enumerations of entities from webpages. Based on an example node a full XPath is constructed from which the indices are removed. In our algorithm we use a similar approach to determine the generalized XPath of a node [39]. Tran et al. propose a method for XPath wrapper induction for extracting product information from webpages. The method uses information about product details in user queries for ranking the XPaths [38]. Zheng et al. propose the use of a 'broom' to extract records from webpages [46]. The broom stick is used as a full XPath to the top of the record region. The broom head describes the DOM tree which contains the record. A major difference to our work is that the system requires manual labeling of records.

3. AUTOMATICALLY FINDING SEARCH RESULT XPATHS

Using XPaths to extract search result records

Our goal is to create an algorithm which, given a single search result webpage, suggests a ranked list of XPaths which can be used to extract the search result records. Ideally, each of the nodes retrieved with the highest ranked XPath completely captures a single SRR.

Figure 1 illustrates the approach. Given a single web search results page from Bing, four XPaths are suggested for extracting SRRs. Using the first XPath, 10 nodes are extracted, each representing a complete SRR. The extracted nodes are highlighted and numbered in the figure. Note that the advertisement above the search results is ignored.

For some webpages it is impossible to completely capture a SRR with a single node. For instance when the search results records are represented by multiple sibling nodes, as in the following example:

```
<dl>
<dt>Title of SRR1</dt>
<dd>Description of SRR1</dd>
<dt>Title of SRR2</dt>
<dd>Description of SRR2</dd>
...
</dl>
```

Even in those cases, a method based on XPaths can be useful for retrieving SRRs. As suggested in [32], the nodes retrieved with an XPath can be used as an *anchor* to locate part of the SRR. The remaining part of the search result can be described relative to this anchor. For instance, `//dt` can be used to retrieve the node with the title of the SRR and serve as an anchor to retrieve the corresponding description with the XPath `./following-sibling::dd[1]`.

Approach

Our overall approach is as follows:

- Generate candidate XPaths based on node repetition and node attributes.
- Rank the candidate XPaths based on a number of features, including the visibility and rendered area of the nodes, the similarity of the nodes and the detection of a grid.

In the following sections these two steps are discussed in more detail.

For flexibility and efficiency we make we make the following assumptions:

1. A single XPath expression can be used to extract (parts of) all SRRs on a search result page. It is convenient to be able to specify the SRR nodes with a single expression.
2. We assume that the search result records are found at the same depth in the DOM tree. Similar assumptions are made in e.g. [27, 46].
3. Only 'basic' XPath expressions are considered: a sequence of parent-child nodes with zero or more predicates.
4. A search result node is either an anchor node (`a` tag) or contains at least one anchor tag.

3.1 Finding candidate XPaths

Finding candidate XPaths is a four-step process.

3.1.1 Find anchor nodes

In the first step all anchor nodes on the page are extracted. These are all nodes specified with the XPath `//a`.

3.1.2 Group nodes based on their generalized XPaths

The nodes found in the previous step are grouped according to their *generalized XPath*. The generalized XPath of a node is defined as the node names encountered when traversing from the document root to that node. For instance, all list item nodes (`li`) found in the example HTML tree in Figure 2, have the generalized XPath `/html/body/div/ul/li`. Only generalized XPaths which retrieve more than a minimal number of nodes (by default three) are kept.

3.1.3 Build predicate tables

For each generalized XPath a *predicate table* is built. The predicate table maintains for each *level* of the generalized XPath the possible predicates and the number of anchor nodes extracted using that predicate. The predicates are based on the node attributes used by the ancestors (of the nodes retrieved by the generalized XPath) at that level. Each node attribute results in a single XPath predicate. For instance, if a node has an attribute `width` with a value `100%`, the predicate `@width='100%'` is added. `id` attributes and `class` attributes are handled differently: the `class` value is split based on whitespace and added as multiple predicates. This is to handle nodes which use a number of style classes simultaneously. If the value of the `id` attribute value ends with a number, these numbers are discarded and the predicate `starts-with(@id, 'value')` is used. This is to capture groups of nodes with a common `id` prefix. In the predicate table, the subset of anchor nodes selected is represented by a nodemask for fast comparison.

Figure 2 shows a simple HTML node tree containing two div nodes and three and ten list (and anchor) nodes found below them respectively. Table 1 lists the corresponding predicate table.

Next, the following type of predicates are removed:

- predicates which select fewer than the minimal number of nodes.

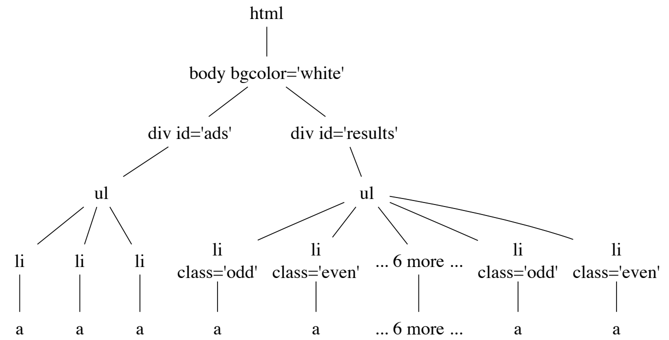


Figure 2: Example HTML tree.

Level	Predicates	# a-nodes	Nodemask
0 /html			
1 /body	`@bgcolor='white'`	13	1111111111111
2 /div	`@id='results'`	10	0001111111111
	`@id='ads'`	3	1110000000000
3 /ul			
4 /li	`@class='even'`	5	0000101010101
	`@class='odd'`	5	0001010101010
5 /a			

Table 1: Example predicate table for for the generalized XPath `/html/body/ul/li/a` which retrieves 13 nodes (corresponding to Figure 2)

- predicates which do not reduce the number of selected nodes.
- predicates which select the same set of nodes as a shorter (measured in the number of characters) predicate—i.e. prefer shorter predicates.

3.1.4 Generate candidate XPaths

Each of the remaining predicates in the predicate table is used to initiate a search for candidate XPaths. For instance, the predicate `@id='results'` in Table 1 results in the XPath `/html/body/div[@id='results']/ul/li/a`.

Subsequently, this XPath is further explored by testing the nodes at a higher level. For the example `/html/body/div[@id='results']/ul/li/a/..` (which can be rewritten as `/html/body/div[@id='results']/ul/li[./a]`), `/html/body/div[@id='results']/ul[./li/a]` etc. are tested. The 'highest' XPath resulting in a distinct number of nodes is added as a candidate XPath.

Finally each of the candidate XPaths is simplified by constructing a relative XPath. First, when available, identifier predicates are added to the XPath. Then the leftmost part is iteratively removed until a larger set of nodes is retrieved by the simplified XPath.

We note that predicates can also be combined, leading to an exponential number of predicate combinations to be used in candidate XPaths. In our experiments we noticed that in most cases an XPath with a single predicate is expressive enough to select the exact set of search result nodes. We therefore limit our candidates to XPaths with a single constraint from the predicate table. Note, however, that more predicates can be added in the simplification process.

3.2 Ranking and filtering candidate XPaths

The set of candidate XPaths is ranked according to the following criteria:

1. The *similarity* of the nodes retrieved by the XPath. SRRs are expected to look similar and there should be no 'outliers': no SRR should be very different from the rest.
2. The *presentation* of the search result nodes. Search results are frequently presented as either a vertical list, where the search results appear below each other. Or, in case of shopping results or image search results, the records are frequently displayed in a grid with a number of rows and columns.
3. The *rendered area* the nodes retrieved by the XPath. The search result nodes are expected to take up a large part of the screen. This requires the webpage to be rendered so width and height can be determined.

In summary: XPaths with a similarity exceeding a similarity threshold and with a grid or row presentation are ranked according to descending rendered area.

Based on initial results of this ranking (and with an early version of the candidate generation process), two additional filtering heuristics are introduced.

The first heuristic handles a preference for XPaths retrieving 'rows' of results. A limitation of this approach is that in case the search results are presented in a grid, an XPath retrieving rows of search results can be preferred over an XPath which retrieves the cells found in the grid. The rows form a coherent group and also its presentation below each other make it a plausible list of search results. Moreover, the rows take up at least as much or more space than the cells found in these rows. The heuristic compares all candidate XPaths and removes an XPath p when an XPath c can be found for which the following constraints hold:

- The number of nodes retrieved by c is a multiple of p: $|c| > 2|p|$
- The nodes retrieved by c form a grid with more than one column.
- Each node retrieved by c is a descendant of a node retrieved by p.
- Each node retrieved by p has at least on descendent retrieved by c.

The second heuristic deals with 'invisible' nodes. Websites may add nodes to the page DOM tree which are not directly visible[5] to the user. These nodes may form, for instance, a dropdown menu and become visible when the user hovers over the navigation menu. This heuristic removes XPaths which retrieve invisible nodes.

In the following subsubsections the similarity calculation and the detection of the presentation are described.

3.2.1 Similarity calculation

The similarity calculation determines two values for a set of nodes retrieved by an XPath: an *average* similarity and a *minimal* similarity. The first value indicates whether on average the nodes are structurally similar. The second value indicates whether there exists a single node which is considerably different from all the other nodes (an outlier).

[5]invisible nodes can be realized with cascading style sheets in several ways: by setting the 'opacity' to zero; setting the 'visibility' to 'hidden'; setting the 'display' to 'none'; or by reducing the width and height to zero.

Dimension	length
`li`	1
`li/p`	3
`li/p/b`	2
`li/p/img`	1

Table 2: Vector representation of a node

The structure of a node is represented as a vector. The dimensions of the vector are relative XPaths to reach the descendants of the node. The size indicates the number of descendant nodes with this relative path.

For example, the node n is represented in HTML as follows:

```
<li>
<p>Title of the <b>result</b></p>
<p>Description of the <b>result</b></p>
<p><img src='image.png'></p>
</li>
```

The structure of n is represented by the vector in Table 2. Note that the length of other dimensions (other relative XPaths) is 0.

Given two vectors v_1 and v_2, representing two nodes, the similarity is calculated using the cosine similarity:

$$sim(v_1, v_2) = \frac{v_1 \cdot v_2}{|v_1||v_2|} \quad (1)$$

where $\cdot$ is the dot product and $|v_1|$ and $|v_2|$ are the lengths of the vectors.

The average similarity could be calculated by averaging all pair-wise node comparisons. However, this calculation would have a complexity of $O(n^2)$ where n is the number of nodes retrieved by an XPath. To reduce the complexity to $O(2n)$, we first calculate a centroid vector v_c by summing the vectors of the retrieved nodes.

$$v_c = \sum_{v \in V} v \quad (2)$$

Where V is the set of vectors representing the nodes retrieved by the XPath and. The average and minimal similarity are then calculated as follows:

$$avgSim = \frac{1}{|V|} \sum_{v \in V} sim(v_c, v) \quad (3)$$

$$minSim = \min_{v \in V} sim(v_c, v) \quad (4)$$

3.2.2 Determining presentation

The presentation of the search result page is determined by analyzing the positions of the nodes on the rendered webpage. In a single pass the top and left coordinates of the rendered nodes are counted. The highest frequency of a top coordinate indicates the number of columns; the highest frequency of a left coordinate indicates the number of rows.

In case the number of nodes retrieved by an XPath does not correspond to the number of nodes you would expect

on the detected grid, the XPath is discarded. In a formula, an XPath retrieving n nodes on a detected grid with r rows and c columns is discarded when $n > (r+1)c$. $r+1$ is used to accommodate for slightly more nodes in the grid.

Also XPaths resulting in a detected grid with only a single row are discarded.

3.3 Parameters

Three parameters are required by the algorithm:

- *minSimilarityThreshold*. The minimum value of the minSimilarity required for a XPath. This value indicates the tolerance for an outlier in a set of result nodes. Based on initial experiments set to 0.55.
- *avgSimilarityThreshold*. The minimum value of the avgSimilarity required for a candidate XPath. This value indicates how similar the set of result nodes should be. Based on initial experiments set to 0.65.
- *minimumNodeCount*. The minimum number of nodes a generalized XPath has to retrieve. Set to 3.

3.4 Implementation and availability

The algorithm is implemented in Javascript running as either a Mozilla Firefox plugin (Figure 1 shows a screenshot) or as a XUL[6] application. The program uses Firefox for retrieving the webpage, constructing the document object model (node tree) and rendering the webpage. The Firefox plugin presents a GUI to highlight the nodes retrieved by the ranked XPaths. The XUL application provides an API to run the algorithm on a webpage. Given a URL, the application returns the highest ranked XPath for that page. The source code is available for download[7].

4. EXPERIMENTAL SETUP

The proposed method is evaluated based on six datasets with web search results. We first describe the datasets used. Then we describe the evaluated aspects.

4.1 Datasets

Six datasets with web search results are used for the evaluation. The first three (*zhao1-3*) were used before in [43]. It contains a total of 246 search result pages found in different domains (general, news, government, medical etc.). The fourth dataset (*yamada*) was assembled by Yamada et al [41] and contains a variety of 51 search result pages. In Yamada's original dataset multiple pages from each search engine are available; in our evaluation we only use a single search result page from each engine. Note that this is a considerably harder task.

We found that these four datasets are limited for a number of reasons. First, these datasets are rather old, making them unrepresentative for the current state of the web. Not only has the layout changed, also the technology has changed. In our experience techniques such as CSS are currently more frequently used than HTML tables. Second, image and video search results are underrepresented in the datasets. Third, the datasets only contain the HTML source of the search result pages making them incomplete for rendering in their original form: images and cascading style sheets were discarded by the creators.

[6]https://developer.mozilla.org/En/XUL
[7]http://www.ewi.utwente.nl/~trieschn/srf/

Testset	#Pages	Number of SRRs			
		Min	Max	Mean	Median
zhao1	97	4	50	14.8	10
zhao2	102	5	437	25.6	10
zhao3	47	3	50	18.7	15
yamada	50	5	160	22.0	10
web1	115	5	999	30.6	11
web2	105	6	100	19.4	15

Table 3: Dataset statistics

Hence, we decided to assemble two additional datasets (*web1-2*). The first has been used to develop the system and was used to train the parameters. The second was assembled separately and only used for the evaluation. For *web1* the search result pages were gathered from the top 500 US websites listed by Alexa[8]. Websites requiring a user account (LinkedIn, facebook), websites containing pornographic material and torrent downloads were discarded. The search function on the main page of the webpage was used to obtain a search result page. The complete result page was downloaded in Mozilla Archive Format[9], which includes all images and CSS files and removes javascript. For *web2* the list of top UK websites listed by Alexa was used in a similar fashion. Websites already used for *web1* were skipped.

For all datasets, we manually determined a single XPath to extract all and the complete SRR. In most cases (87 %) this was possible. For 13% of the pages it was not possible to use a single XPath to retrieve the *complete* SRR; for instance when all titles and descriptions are sibling rows in the same table. In these cases only the title (or image, in case of image/video search) was selected.

We note that in some cases determining the set of SRRs of a page is debatable. In all cases we left out ads in the desired set of SRRs. In case of multiple result sections, we chose the in our opinion most important section.

The characteristics of the datasets are listed in Table 3. The datasets and ground truth can be downloaded from the website mentioned before.

4.2 Evaluation aspects

We evaluate our method on a number of aspects.

First, the most important evaluation aspects is *accuracy*. Does the highest ranked XPath retrieve the complete and exact set of manually annotated SRRs on a page? This can be achieved by comparing the nodes retrieved by the highest ranked XPath with the manually determined XPath. However, this can be deceitful since two different nodes can retrieve almost exactly the same SRR. Consider, for instance, a single SRR which is enclosed by a table containing a single row. Both the table node and the row node retrieve the complete search result. In our evaluation, we therefore compare a *text-only representation* of the nodes. This representation consists of the text found below a node and the values of alternative-text and src attributes. We classify the accuracy of the method for a particular search result page in 4 categories: *perfect*, when all the intended SRR-nodes are precisely retrieved, *too many* and *too few* when at least

[8]http://www.alexa.com/topsites
[9]http://maf.mozdev.org/

one intended node is precisely retrieved but too many or too few nodes are returned, and *incorrect* when one of the previous does not apply. We think that such a classification is more insightful than micro or macro precision, recall and F-measure, which can easily mask small errors by averaging over a large set of SRRs. We carry out an error analysis to determine what causes the errors.

Second, we determine the contribution of the various heuristics used in the algorithm. Using a heuristic might aid in solving a problem on one page, but might cause an error on another. We would have liked to compare our algorithm to an existing algorithm, but to the best of our knowledge no existing systems give comparable output (i.e. given a single webpage returns an XPath to retrieve SRRs). We therefore carry out this breakdown analysis of the method.

Third, we analyze the running time of the algorithm. The complexity of the major components of the algorithm is linear to the number of nodes and attributes in the HTML page, which can be considerably different across pages. The running time statistics give a clear impression of the speed of the algorithm.

5. RESULTS

5.1 Accuracy

Table 4 lists the accuracy of the algorithm for the six datasets. As expected, the performance on the *web1* dataset is highest since this set was used to develop the algorithm. For 84 webpages (73%) an XPath is suggested which precisely and completely matches the SRRs in the ground truth. For 4 pages (3%), all ground truth SRRs are extracted, plus one or more incorrect SRRs; for 15 pages at least some ground truth SRRs are extracted and for 12 pages only incorrect SRRs are extracted. For *web2*, the accuracy is slightly lower (67% precisely correct). The percentage of correctly suggested XPaths for the other and older testsets varies between 53% and 63%. The test sets *zhao1-3* and *yamada* show a relatively high number of pages where too many SRRs are extracted. In contrast, *web1-2* have more pages where too few SRRs are extracted.

Table 5 provides more insight into the errors made. For each page with too many, too few or incorrectly extracted SRRs we determined the cause of the error. Moreover, we judged whether a (limited but) *useful* set of SRRs was extracted. We judged the extracted sets as useful when 1) next to *all* SRRs, up to three additional rows were included; 2) when 80% or more of the SRRs was found and only correct SRRs were retrieved and 3) when a part of all SRRs was retrieved which could be used as an anchor. Note that in Table 5 the errors of all test sets were aggregated. Figure 3 illustrates four types of causes. We will limit the following discussion to the most frequent causes. In most of the cases when too many SRRs were extracted, the additional records were navigational rows directly above or below the actual search results. This happened particularly often with the older test sets, which contain relatively many tables for presenting the results. Most of the extracted nodes are still useful as search results. In four cases the SRRs were 'doubled': in those cases each SRR is represented by multiple nodes. The cases where too few result nodes are retrieved show a limitation of the bottom-up approach the algorithm takes. Since links (anchor tags) are used as a starting point for finding SRRs, this approach can easily make an incomplete selection of SRRs of a particular type. For instance, by only selecting the SRRs which have a cache-link, or a link to a product with contact information. Since groups of similar nodes are preferred, XPaths also including SRRs of a special type (e.g. movies, news or images) can be discarded. In most of those cases (26 of 27) the nodes which are extracted are still useful as search results. Another type of errors is caused by an irregular tree structure: the first result is then found at a different level than the other search results. Also indented search results (typically indicating a near duplicate or closely related result) are sometimes missed. In both cases the remaining search results are complete and correct. Missing SRRs of a special type or missing indented SRRs is frequently observed in the *web* test sets. The results illustrate that modern search engines more frequently present blended aggregated search results (e.g. images and news mixed with web results) and use indentation in their result presentation. Finally, we analyzed the pages which were initially labelled incorrect. Most of the incorrect results are pages in which 'double' are detected. A typical example is a table with each row representing a SRR. The algorithm incorrectly identifies two or more result cells in each row. Another large group of pages is labelled as incorrect because not the title but a larger node (linked to the correct SRR) is preferred over the title. Such nodes can still be used as useful anchors to extract the complete SRR.

Accuracy / Cause	Count	Useful
too many (50)		
up to 3 additional rows (header, navigation, ad)	46	46
double results	4	
too few (46)		
missed special type	14	14
missed SRRs lacking attribute	13	12
irregular tree structure	8	8
missed indented result	5	5
multiple SRRs as single node	3	2
many incorrect	2	
prefers larger area	1	1
incorrect (89)		
double results	20	
description over title	12	12
unclear	10	
prefers blocks of results	9	
prefers larger repetitive parts	7	
nested tree structure	5	
prefers larger area	5	
undetected hidden elements	4	
no strict grid	4	
ground truth problem	3	3
similarity	4	1
irregular structure	3	3
missed SRRs lacking attributes	2	2
no anchors in SRRs	1	
total	185	109

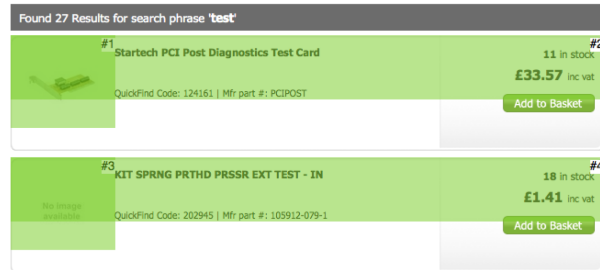
Table 5: Error analysis of pages from which SRRs were not completely and precisely extracted

Table 5 puts the modest correct percentages of Table 4 in perspective: many (109 out of 185) of the not-correct results are still useful for extraction. On average (over all test sets) this would result in useful XPaths for 85% of the pages.

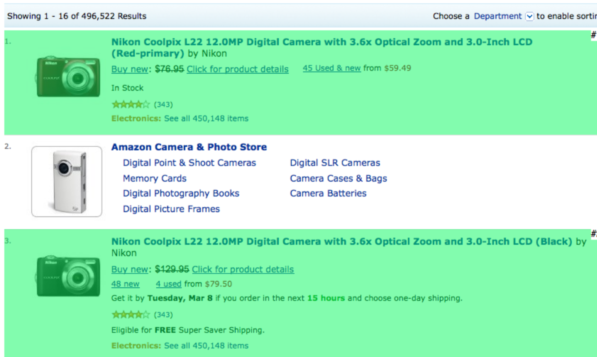
Result	zhao1		zhao2		zhao3		yamada		web1		web2	
correct	63	65%	54	53%	28	60%	32	64%	84	73%	70	67%
too many	13	13%	16	16%	5	11%	5	10%	4	3%	7	7%
too few	5	5%	5	5%	6	13%	4	8%	15	13%	11	10%
incorrect	16	16%	27	26%	8	17%	9	18%	12	10%	17	16%

Table 4: Accuracy of the method on six datasets

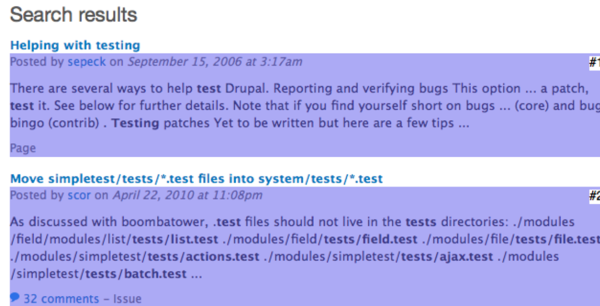
(a) up to 3 additional rows



(b) double results



(c) missed special type



(d) description over title

Figure 3: Screenshots of some typical non-correct cases (see Table 5)

5.2 Heuristics

Table 6 lists the impact of the various heuristics implemented in the algorithm. Each row shows the change in the correct number of pages. A positive value indicates that turning the heuristic off leads to more correctly extracted pages. The percentage indicates the fraction of the number of correct pages.

The heuristic of removing XPaths retrieving invisible nodes turns out to have only little effect. The manual analysis already showed that in some cases invisible nodes were not detected. In early versions of the algorithm, which used a topdown approach to select nodes which contain links, the heuristic was observed to be more important.

Unexpectedly, the heuristic of removing rows of results turns out to hurt the performance more than it helps: for all test sets the number of correctly extracted pages increases with the heuristic turned off. For the *zhao* testsets, the heuristic only has a negative effect. This can be explained by the fact that these testsets contain many results in tabular format (fixing many of the "double results" mentioned in Table 5). For the other test sets some pages improve, but more are hurt by the heuristic.

The other three heuristics do show a positive improvement of the results.

Using the predicate table has the strongest positive impact on the *web* test sets. This can be explained by the fact that these pages depend more on the usage of cascading style sheets and classes. Using these as predicates in XPaths for SRR extraction is useful.

The detection of a grid (and the requirement of multiple rows and number of nodes approximating the nodes in the grid), also turns out to have a positive effect on SRR extraction. Especially the *web* test sets benefit from these heuristics. An explanation is that these results contain many results in grid form, such as image and shopping result pages, which are more frequently shown in a grid.

The re-ranking based on similarity shows to have the largest positive impact on performance, leading to up to 17 more correctly extracted pages in comparison to not using the heuristic.

5.3 Running time

Table 7 shows the running time statistics of the algorithm on the test sets. It should be noted that the algorithm has

Heuristic	zhao1		zhao2		zhao3		yamada		web1		web2	
remove invisible nodes	1	2%	0	0%	0	0%	0	0%	-1	-1%	0	0%
remove rows of results	2	3%	5	9%	2	7%	1	3%	2	2%	7	10%
use predicate table	-3	-5%	0	0%	0	0%	-1	-3%	-9	-11%	-2	-3%
grid detection	-3	-5%	-1	-2%	0	0%	0	0%	-6	-7%	-10	-14%
similarity filtering/ranking	-5	-8%	-8	-15%	-1	-4%	-3	-9%	-17	-20%	-13	-19%

Table 6: Change in correct number of web pages with individual heuristics turned off (a negative value indicates the heuristic improves the results)

	Min	Max	Mean	Median
zhao1	54	6,309	621	388
zhao2	34	11,930	967	327
zhao3	49	2,995	527	428
yamada	40	4,784	718	346
web1	161	58,695	2,752	1,285
web2	133	63,925	4,398	1,819

Table 7: Running time per page in milliseconds

been implemented in Javascript and probably could be further optimized. Tests were performed running in the background on a 2.2 GHz laptop with 2 GB RAM. On average, processing a page takes up to 4.4 seconds. The outlier (taking the algorithm over a minute to complete) is a page with over 1300 links in a very deep hierarchical structure. The 1300 links result in many deep generalized xpaths which are further explored for candidate XPaths. The algorithm could be made more efficient by pruning generalized XPaths which retrieve a set of nodes covering only a small area of the screen. A typical (median) page takes between 327 miliseconds and 1.8 seconds, which we consider acceptable. From the running time statistics we conclude that current web search results have become increasingly large and complex.

6. DISCUSSION

The evaluation shows that retrieving SRRs can be achieved using automatically determined XPaths. In this section we will discuss the limitations of the work presented here.

Some caveats can be encountered when using the presented approach in practice. The method uses the Document Object Model (DOM) generated by the parser. In case of real-world (dirty) HTML pages, constructing the DOM is not trivial; different HTML parsers may repair incorrect HTML in different ways, potentially leading to different DOM trees, resulting to different XPaths from our algorithm. This would make the approach using XPaths less portable than we intended. The fix to this problem is to make sure the same parser is used for finding the XPaths and when later using the XPaths for extracting. When the pages are compliant with the HTML standard no problems should occur.

A limitation of this work is that we have not evaluated the reusability of the extracted XPaths for other search result pages from the same template. We will carry out this evaluation in future work. We did notice that the current algorithm has a preference for XPaths which strongly rely on structure rather than on attribute predicates. We think this explains the only limited impact of using the predicate table: a (set of) node(s) turns out to be precisely specified by only the relative structure of its parent and child nodes. What these XPaths look like might influence its reusability on pages from the same template. This could extend recent work from [15], who proposed a probabilistic model for finding robust XPath extractors.

The evaluation of the contribution of the individual heuristics shows that devising such heuristics is far from trivial. A heuristic which solves a problem on one page can easily introduce more problems on other pages. An interesting direction for future work would be to devise a number of potentially useful heuristics and finding an optimal combination based on machine learning.

7. CONCLUSION

In this work we have proposed an automatic approach to suggest XPaths for extracting search result records from web search result pages. We have evaluated the approach on a number of old and two new test sets. The results on the test sets show that search result pages have become increasingly complex; there is a large variety between search result displays. Moreover, individual search result pages have become complex with for example aggregated search results. XPaths can be effectively used to extract search result records, either by extracting the search result as a whole, or by providing an anchor to part of the record. The algorithm presented here suggests precise and correct XPaths for up to 74% of the pages in the test sets. Overall, for 85% of the search result pages, useful search results can be extracted with the suggested XPaths. The algorithm and used datasets are available for download for follow-up research[10].

Acknowledgements

We thank Robin Aly for his insightful suggestions. This research was supported by the Netherlands Organization for Scientific Research, NWO, grant 639.022.809.

References

[1] B. Adelberg. NoDoSE - a tool for semi-automatically extracting structured and semistructured data from text documents. In *SIGMOD '98*, pages 283–294, 1998.

[2] M. Álvarez, A. Pan, J. Raposo, F. Bellas, and F. Cacheda. Extracting lists of data records from semi-structured web pages. *Data & Knowledge Engineering*, 64(2):491–509, 2008.

[3] T. Anton. XPath-wrapper induction by generalizing tree traversal patterns. *Lernen, Wissensentdeckung und Adaptivitt (LWA)*, pages 126–133, 2005.

[10] `http://www.ewi.utwente.nl/~trieschn/srf`

[4] A. Arasu and H. Garcia-Molina. Extracting structured data from web pages. In *SIGMOD '03*, pages 337–348, 2003.

[5] G. Arocena and A. Mendelzon. WebOQL: Restructuring documents, databases and webs. In *14th International Conference on Data Engineering*, pages 24–33, 1999.

[6] R. Baumgartner, S. Flesca, and G. Gottlob. Visual web information extraction with lixto. In *VLDB '01*, pages 119–128, 2001.

[7] D. Buttler, L. Liu, and C. Pu. A fully automated object extraction system for the world wide web. In *21st International Conference on Distributed Computing Systems*, pages 361–370, 2001.

[8] C. Chang, M. Kayed, M. Girgis, and K. Shaalan. A survey of web information extraction systems. *IEEE transactions on knowledge and data engineering*, pages 1411–1428, 2006.

[9] C. Chang and S. Lui. Iepad: information extraction based on pattern discovery. In *WWW '01*, pages 681–688, 2001.

[10] B. Chidlovskii, J. Ragetli, and M. de Rijke. Automatic wrapper generation for web search engines. In H. Lu and A. Zhou, editors, *Web-Age Information Management*, volume 1846 of *Lecture Notes in Computer Science*, pages 399–410. Springer Berlin / Heidelberg, 2000.

[11] E. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[12] V. Crescenzi and G. Mecca. Grammars have exceptions. *Information Systems*, 23(8):539–565, 1998.

[13] V. Crescenzi and G. Mecca. Automatic information extraction from large websites. *Journal of the ACM*, 51:731–779, September 2004.

[14] V. Crescenzi, G. Mecca, P. Merialdo, et al. Roadrunner: Towards automatic data extraction from large web sites. In *VLDB '01*, pages 109–118, 2001.

[15] N. Dalvi, P. Bohannon, and F. Sha. Robust web extraction: an approach based on a probabilistic tree-edit model. In *SIGMOD '09*, pages 335–348, 2009.

[16] N. Dalvi, R. Kumar, and M. Soliman. Automatic wrappers for large scale web extraction. *VLDB Endowment*, 4:219–230, January 2011.

[17] D. Embley, Y. Jiang, and Y. Ng. Record-boundary discovery in web documents. In *SIGMOD '99*, pages 467–478, 1999.

[18] D. Freitag. Information extraction from html: Application of a general machine learning approach. In *National Conference on Artificial Intelligence*, pages 517–523, 1998.

[19] D. Freitag. Multistrategy learning for information extraction. In *Fifteenth International Conference on Machine Learning*, pages 161–169, 1998.

[20] J. Hammer, J. McHugh, and H. Garcia-Molina. Semistructured data: The TSIMMIS experience. In *First East-European Workshop on Advances in Databases and Information Systems*, pages 1–8, 1997.

[21] J. Hong, E. Siew, and S. Egerton. Information extraction for search engines using fast heuristic techniques. *Data & Knowledge Engineering*, 69(2):169–196, 2010.

[22] C. Hsu and M. Dung. Generating finite-state transducers for semi-structured data extraction from the web. *Information systems*, 23(8):521–538, 1998.

[23] S. Kuhlins and R. Tredwell. Toolkits for generating wrappers. *Objects, Components, Architectures, Services, and Applications for a Networked World*, pages 184–198, 2009.

[24] A. Laender, B. Ribeiro-Neto, A. da Silva, and J. Teixeira. A brief survey of web data extraction tools. *ACM Sigmod Record*, 31(2):84–93, 2002.

[25] M. Lam, Z. Gong, and M. Muyeba. A method for web information extraction. *Progress in WWW Research and Development*, pages 383–394, 2008.

[26] K. Lerman, C. Knoblock, and S. Minton. Automatic data extraction from lists and tables in web sources. In *IJCAI-2001 Workshop on Adaptive Text Extraction and Mining*, volume 98, 2001.

[27] B. Liu, R. Grossman, and Y. Zhai. Mining data records in Web pages. In *SIGKDD '03*, pages 601–606, 2003.

[28] K. Liu, W. Meng, J. Qiu, C. Yu, V. Raghavan, Z. Wu, Y. Lu, H. He, and H. Zhao. Allinonenews: development and evaluation of a large-scale news metasearch engine. In *SIGMOD '07*, pages 1017–1028, 2007.

[29] W. Liu, X. Meng, and W. Meng. Vide: a vision-based approach for deep web data extraction. *IEEE Transactions on Knowledge and Data Engineering*, pages 447–460, 2009.

[30] F. McCown and M. L. Nelson. Search engines and their public interfaces: which apis are the most synchronized? In *WWW '07*, pages 1197–1198, 2007.

[31] G. Miao, J. Tatemura, W.-P. Hsiung, A. Sawires, and L. E. Moser. Extracting data records from the web using tag path clustering. In *WWW '09*, pages 981–990, 2009.

[32] J. Myllymaki and J. Jackson. Robust web data extraction with xml path expressions. *IBM Research Report*, 23, 2002.

[33] N. Papadakis, D. Skoutas, K. Raftopoulos, and T. Varvarigou. Stavies: A system for information extraction from unknown web data sources through automatic web wrapper generation using clustering techniques. *IEEE Transactions on Knowledge and Data Engineering*, pages 1638–1652, 2005.

[34] A. Sahuguet and F. Azavant. Building intelligent web applications using lightweight wrappers. *Data & Knowledge Engineering*, 36(3):283–316, 2001.

[35] K. Simon and G. Lausen. Viper: augmenting automatic information extraction with visual perceptions. In *CIKM '05*, pages 381–388, 2005.

[36] S. Soderland. Learning to extract text-based information from the world wide web. In *KDD '97*, pages 251–254, 1997.

[37] W. Su, J. Wang, and F. Lochovsky. ODE: Ontology-assisted data extraction. *Transactions on Database Systems*, 34(2):12, 2009.

[38] N. Tran, K. Pham, and Q. Ha. XPath-wrapper induction for data extraction. In *International Conference on Asian Language Processing, IALP '10*, pages 150–153, 2010.

[39] D. Urbansky, M. Feldmann, J. Thom, and A. Schill. Entity extraction from the web with WebKnox. *Advances in Intelligent Web Mastering-2*, pages 209–218, 2010.

[40] J. Wang and F. H. Lochovsky. Data extraction and label assignment for web databases. In *WWW '03*, pages 187–196, 2003.

[41] Y. Yamada, N. Craswell, T. Nakatoh, and S. Hirokawa. Testbed for information extraction from deep web. In *WWW Alt. '04*, pages 346–347, 2004.

[42] Y. Zhai and B. Liu. Web data extraction based on partial tree alignment. In *WWW '05*, pages 76–85, 2005.

[43] H. Zhao, W. Meng, Z. Wu, V. Raghavan, and C. Yu. Fully automatic wrapper generation for search engines. In *WWW '05*, pages 66–75, 2005.

[44] H. Zhao, W. Meng, and C. Yu. Automatic extraction of dynamic record sections from search engine result pages. In *VLDB '06*, pages 989–1000, 2006.

[45] S. Zheng, M. R. Scott, R. Song, and J.-R. Wen. Pictor: an interactive system for importing data from a website. In *SIGKDD '08*, pages 1097–1100, 2008.

[46] S. Zheng, R. Song, J.-R. Wen, and C. L. Giles. Efficient record-level wrapper induction. In *CIKM '09*, pages 47–56, 2009.

[47] J. Zhu, Z. Nie, J. Wen, B. Zhang, and W. Ma. Simultaneous record detection and attribute labeling in web data extraction. In *SIGKDD '06*, pages 494–503, 2006.