Structure-Driven Crawler Generation by Example

Márcio L. A. Vidal Altigran S. da Silva Edleno S. de Moura João M. B. Cavalcanti

Universidade Federal do Amazonas
Departamento de Ciência da Computação
Manaus – Brazil
{mvidal,alti,edleno,john}@dcc.ufam.edu.br

ABSTRACT

Many Web IR and Digital Library applications require a crawling process to collect pages with the ultimate goal of taking advantage of useful information available on Web sites. For some of these applications the criteria to determine when a page is to be present in a collection are related to the page content. However, there are situations in which the inner structure of the pages provides a better criteria to guide the crawling process than their content. In this paper, we present a structure-driven approach for generating Web crawlers that requires a minimum effort from users. The idea is to take as input a sample page and an entry point to a Web site and generate a structure-driven crawler based on *navigation patterns*, sequences of patterns for the links a crawler has to follow to reach the pages structurally similar to the sample page. In the experiments we have carried out, structure-driven crawlers generated by our new approach were able to collect all pages that match the samples given, including those pages added after their generation.

Categories and Subject Descriptors

H.3 [**Information Systems**]: Information Storage and Retrieval—*Clustering*;

General Terms

Algorithms, Experimentation

Keywords

Digital Libraries, Tree Edit Distance, Web Crawlers

1. INTRODUCTION

An ever increasing number of applications on the Web target at processing collections of similar pages obtained from Web sites. The ultimate goal is to take advantage of the valuable information these pages implicitly contain to perform tasks such as creating digital libraries, querying, searching, data extraction, data mining and feature analysis. For some of these applications, the criteria to determine when a page is to be present in a collection are related to the page content, i.e. words, phrases, etc. However, there are other situations in which the features of the inner structure of the pages provide better criteria to define a collection than their content.

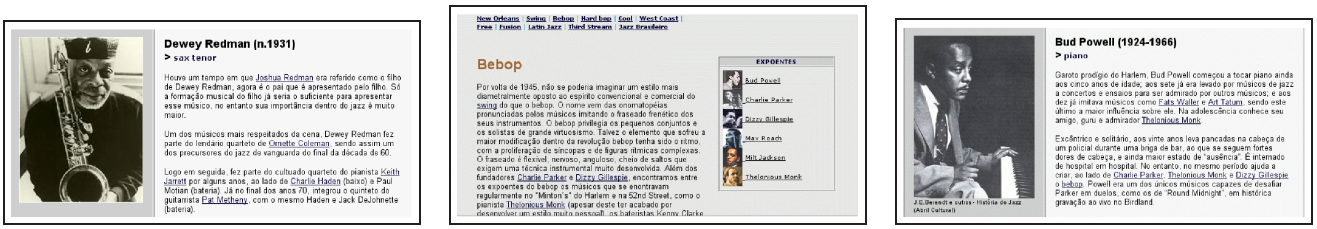
For instance, consider an application that requires collecting pages containing information about Jazz artists whose pages are available in the *E-Jazz* Web Site[1]. To define a set of content-related features that encompasses all the artists would be a hard task, since artists are usually related to some jazz genre or to a musical instrument, and there will be several distinct styles and instruments to be considered. On the other hand, there will also be non-artist related pages that share a same subject with several artist pages. For instance, the page in Figure 1(a) is an artist page that shares the same subject with the page in Figure 1(b), which is a Jazz genre page. Furthermore, the page in Figure 1(a) shares no common subject with the page in Figures 1(c), despite the fact that both of them are artist pages. In such situations using content-related features to characterize the pages to be collected is likely to fail.

In such cases, representing the pages to be collected using features related with content is not appropriate. Motivated by this problem, we propose a new Web crawling technique that is based on the structure of the Web pages instead of their content. While many work in the literature have addressed the issue of content-driven Web crawling, the use of page structure as a criterion to guide the traversal of crawlers has been almost neglected. Nevertheless, this is an interesting alternative to solve practical problems in several important Web applications. The applications we consider include: automatically providing data-rich Web pages for wrappers which generally rely on structural patterns for performing extraction of implicit data [3, 12, 19]; structure-aware Web page searching and querying [2, 10]; building of digital libraries [4, 16]; Web usage mining [8] and other applications where a content-driven crawler is not the optmal choice.

In this paper we present an approach for generating structure-driven crawlers that requires a minimum effort from users, since it relies on a handful of examples (usually one) of the pages to be fetched. To accomplish this, given a sample page and an entry point to a Web site, the tool greedily traverses the Web site looking for *target* pages, i.e., pages that are structurally similar to the sample page. Next, it records all paths that lead to target pages and generates a *navigation pattern* [13] which is composed by sequences of patterns of links a crawler has to follow to reach the target

[1] `http://www.ejazz.com.br`

SIGIR'06, August 6–11, 2006, Seattle, Washington, USA.



(a) (b) (c)

Figure 1: Sample pages from the e-jazz Web site.

pages. Finally, we automatically generate a crawler based on these patterns. From this point on, the crawler can be used to fetch pages that are structurally similar to the sample page, even if new similar pages are added later.

A remarkable feature of the structure-driven approach is that in many applications the structural criteria are the only option for producing an effective crawler. Furthermore, the structure criteria is usually more precise and safer than content criteria. As a result, while content-driven crawlers seldom achieve higher quality levels, the structure-driven approach tends to be extremely precise without losing information. Indeed, in the experiments we have carried out involving several crawling sessions over 11 real Web sites, structure-driven crawlers generated by our approach were able to collect almost all pages that matched the samples given, including those pages added long after their generation, achieving 100% precision and at least 95% recall in all cases.

The remainder of the paper is organized as follows. Section 2 covers research initiatives related to ours. Section 3 reviews the concept of tree edit distance and the algorithm we use to evaluate the structural similarity of Web pages. In Section 4 we focus on the details of the structure-driven crawling approach and the techniques used for generating structure-driven Web crawlers. Section 5 describes the results of experiments. Finally, in Section 6 we present our conclusions and suggestions for future work.

2. RELATED WORK

The work we present in this paper is motivated by previous work on automatic Web crawling for feeding wrappers with data-rich pages. More specifically, in [13] the concept of Navigation Pattern *NP* was introduced as a way of guiding the navigation of a crawler through a target Web site to fetch pages of interest for data extraction tasks. The main novelty we introduce in the present paper is that, while in that previous work NPs are fixed and manually generated, our approach automatically generates NPs and the role of the user is limited to providing examples of the pages to be fetched.

The internal structure of the DOM trees associated to Web pages was previously used in [9] and [12] to create clusters of pages that are structurally similar. Like this paper, the main motivation of both work is to organize the pages to feed data extraction processes performed by wrappers. In the present paper, however, we deal with a related but different problem: how to collect pages of a single class of structurally similar pages according to a sample page given as input.

Another noteworthy distinction between this article and previous work is the way the structural similarity between pages is evaluated. In [9] the authors rely on the layout properties of link collections available in the pages. More precisely, they use the set of paths between the root of the page and the HTML tags `<a>` to characterize the structure. Pages that share the same paths are considered structurally similar. Here we adopt a tree edit distance algorithm proposed in [12] called *RTDM (Restricted Top-Down Mapping)*. An adaptation of this algorithm is deployed in order to verify which pages are to be collected by the crawler based on their structural similarity with the given sample page.

The present work is also related to the work on *focused crawling* [6]. In focused crawling the main idea is to have crawlers that only fetch Web pages that are relevant to a given topic. To specify this topic, a set of example Web pages is provided. To drive the crawl process through the Web, a classifier is used to evaluate the relevance of the pages found with respect to the focus topics. We refer to this kind of focused crawler as *content-driven*, since the classifier considers the similarity between the content of the pages found during the crawl and the content of the page given as example. Distinctly, the crawlers addressed in the present paper are termed *structure-driven*, since they rely on the structural similarity of pages.

The use of features other than page content have been considered in recent papers, such as [1, 14]. Among the features taken as evidence for determining how relevant a page is to be fetched are the tokens present in URLs, the nature of Web pages that refer to a given page, and the linking structure of the sites. However, in none of these work the internal structure of the pages was considered as a piece of evidence of similarity.

It is important to notice that in the present paper we deal with a problem that is slightly distinct from the one addressed by typical focused crawlers work. While focused crawlers seek to crawl the whole Web, or a region of interest on the Web, we aim at generating crawlers that continuously fetch pages from specific Web sites known in advance according to an implicit set of structural criteria.

3. STRUCTURAL SIMILARITY

In this section we review the concept of tree edit distance (TED) [17], which is a key concept in our approach and describe how it is used to compare two given Web pages according to their structure. Intuitively, the edit distance between two trees T_A and T_B is the cost associated with the minimal set of operations needed to transform T_A into T_B. Thus, in general, we consider that the similarity between T_A and T_B is inverse to their edit distance.

In our work we assume that the structure of a Web page can be described by a tree, in particular a *labeled ordered rooted tree*. A rooted tree is a tree whose root vertex is fixed. Ordered rooted trees are rooted trees in which the relative order of the children is fixed for each vertex. Labeled ordered

rooted trees have a label attached to each of their vertex.

In its traditional formulation, the TED problem considers three operations: (a) vertex removal, (b) vertex insertion, and (c) vertex replacement. To each of these operations, a cost is assigned. The solution of this problem consists of determining the minimal set of operations (i.e., the one with the minimum cost) to transform one tree into another.

In our work we consider a restricted top-down version of the TED problem which restricts the removal and insertion operations to take place only in the leaves of the trees [7]. Such a formulation is particularly useful for dealing with the HTML pages represented by their underlying DOM trees.

In our approach for generating structure-driven crawlers we adopt an algorithm called RTDM [12], to calculate the tree edit distance between two pages according the restricted top-down formulation. To determine the restricted top-down TED between two trees T_1 and T_2, the *RTDM* algorithm first finds all identical sub-trees that occur at the same level of the input trees. This step is performed in linear time using a graph of equivalence classes. However, *RTDM* is based on a post-order traversal of the trees. Although much simpler, this approach is applicable because we only look for the identical sub-trees of the *same level*. This first step of the algorithm has linear cost, with respect to the number of vertices in the trees. Once the vertices in the trees are grouped in equivalence classes, an adaptation of Yang's algorithm [18] is applied to obtain the minimal restricted top-down mapping between the trees. This second step of the algorithm has a worst case complexity of $O(n_1 n_2)$, but, in practice, it performs much better due to the fact that it only deals with restricted top-down sets of operations. For more details on the *RTDM* algorithm we refer the interested reader to [12].

4. CRAWLER GENERATION

In this section we detail the process of semi-automatically generating structure-driven crawlers proposed in our approach. We begin by illustrating the process by means of a simple example and then we proceed to discuss further details on the two phases involved in it: *site mapping* and *navigation pattern generation*.

4.1 Overview and a Simple Example

Throughout this section, we illustrate some of the concepts we use by taking the *E-Jazz* Web site as an example. The site features information on jazz styles, instruments, recordings and artists. We concentrate on the artist-related portion. The overall structure of the site is presented in a simplified form in Figure 2.

In Figure 2, pages are labeled for reference in the discussion that follows. Assume that all pages labelled $0/2/i$ $(1 \leq i \leq k)$ are similar to page $0/2/0$, i.e., they all contain links to artists pages. We call these pages, *artist hub pages*. Also, assume that all pages labelled $0/2/i/j$ $(1 \leq i \leq K, 1 \leq j \leq K_i)$ are similar to page $0/2/0/0$, i.e., they are all *artist pages*.

Now, suppose we want to fetch all artist pages available on the *E-Jazz* Web site. To accomplish this task, a crawler would begin its traversal at page $0/2$. Next, the crawler would have to access all artist hub pages and finally, it would fetch all artist pages.

Notice that new artist pages can be added to the Web site. This implies that some artist hub pages are often updated

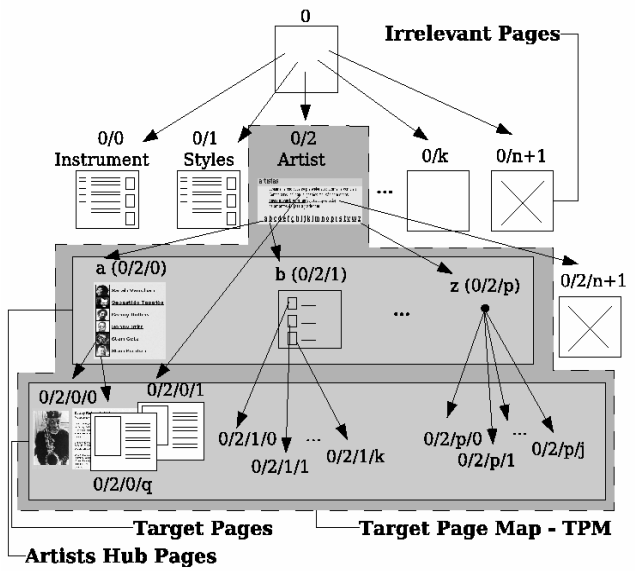


Figure 2: Overall structure of the E-Jazz Web site

to include new links to these new pages. Thus, these new links and pages will have to be taken into account in future crawling processes.

Two parameters are required as input to generate a crawler capable of accomplishing such task: an URL indicating a sample page to serve as an example for the pages to be fetched and another URL indicating an entry point to the site where these pages can be found. In our example, page $0/2/0/0$ serves as the sample page, while page $0/2$ serve as the entry point.

The creation of a crawler is accomplished in two phases: *site mapping* and *navigation pattern generation*. In the site mapping phase, all paths starting from the entry point are traversed to look for target pages. A page is considered as a target when it is structurally similar to the given sample page. Notice that, the traversing is limited to the same Web site. Every path from the entry point that leads to a target page is recorded. Thus, if we consider a Web site as a graph, the site mapping phase generates as output a *minimum spanning tree* where all leaves are target pages. This tree is called *Target Pages Map (TPM)*. In Figure 2 the output of the mapping corresponds to the tree in the area delimited with a dashed line.

In the second phase, navigation pattern generation, the goal is to create a generic representation of the TPM. This generic representation is a list of regular expressions, where each regular expression represents the links occurring in a page the crawler has to follow to reach the target pages. This generic representation is called a *Navigation Pattern (NP)*. A NP is meant to guide a crawler in the traversal of a site to fetch the target pages. Thus, it corresponds to a single path into the TPM. However, as many paths leading to target pages can exist, we choose the one that potentially leads to the greatest number of target pages. Notice that, if correctly generated, regular expressions in the NP will meet our requirement of accounting for new links added to the pages in the site, even after the mapping phase has been completed, since it uses regular expressions to identify links that lead to target pages.

As an example, Table 1 shows a very simple but real navigation pattern for the E-Jazz Web site. In this table, ex-

pression E_1 is applied to the entry page. It matches only links that lead to artist hub pages. Next, the expression E_2 is applied to each artist hub page and matches all the links in these pages that lead to all the artist pages and only to these pages. Notice that no user intervention other than providing the sample page and the entry point is required.

In the following sections we detail the two phases required by our structure-driven approach for producing a structure-driven crawler.

4.2 Site Mapping

In the site mapping phase, we consider a Web site as a directed graph $S = \langle P, L \rangle$, where P is the set of pages in the Web site S and L is a set of pairs $\langle x, y \rangle$ such that a link exists from page x to page y in S. The ultimate goal of this phase is to generate a tree called *Target Pages Map (TPM)* defined as follows.

Definition 1. *Let S be a Web site as defined above. Let $e, p \in S$ respectively be the entry point and the sample page supplied by the user. We define a* Target Pages Map (TPM) *as a tree T that is a subset of the minimum spanning tree of S where e is the root and the set of leaf nodes is formed exactly by the set of nodes p_i from S that are structurally similar to p.*

To generate a TPM we simply crawl the Web site starting from the entry point using a breadth-first traversal procedure. This procedure is described in Figure 3. It takes as input an entry point e of a Web site and a sample page p to be used for comparison. Initially, in Line 7 all links in e are enqueued in Q. Next, the crawling process goes on with the procedure iterating over this queue in the loop of Lines 8–15. The breadth-first traversal is the best choice for mapping the target Web site, since this strategy is known to provide a better coverage of the sites traversed [15].

This iteration ends when the queue is empty or the **Stop** function returns a true value. This function encodes a set of practical constraints for the crawling, for instance, the maximum number of pages fetched or the maximum number of levels to be reached. In our current implementation these constraints are set by means of user-defined parameters, which are fairly simple to be estimated. Notice that these parameters are used only for the site mapping phase and are not required once the resulting crawler is generated.

In Lines 9 and 10 the first element in the queue is taken and compared with p using the **RTDMSim** function. This comparison, is based on the structure of these pages, i.e., we verify how similar their underlying DOM trees are using the RTDM algorithm presented in Section 3. If the current page x and the sample page p are considered structurally similar, we add x to a set X of target pages (Line 11), i.e., pages that must be collected. If x and p are not similar, the links in x are enqueued (Line 12) and the crawling goes on. At the end of the iteration, the set X will have all target pages found stored in it.

After that, in the iteration of Lines 16–19 all nodes and arcs present in the paths from the entry point to target pages in X will be added to the target page map (TPM) T. Notice that the *Mapping* procedure obtains a sub-tree of the minimal spanning tree of the site.

In our running example, the TPM generated would correspond to the tree delimited by the shaded area in Figure 2. Notice that page labeled 0/2/0/1 is directly linked by an

```
1  Mapping
2  input: An entry point e of a Web site S
3         and a sample page p
4  output: A TPM T
5  begin
6    let Q be queue
7    Q ← links extracted from e
8    while Q ≠ ∅ || Stop
9          x ← Dequeue(Q)
10         if RTDMSim(p,x)
11           then X ← X ∪ {x}
12            else Q ← Q ∪ links extracted from x
15   end
16   foreach x ∈ X do
17     let C = ⟨e, v1, ..., vk, x⟩ be the path from e to x
18     Add the nodes and arcs in C to T.
19   end
20 end
```

Figure 3: Procedure used in the site mapping phase.

entry point page 0/2, while all other similar pages near by are linked by the artist hub page labeled 0/2/0. Actually, there is a link in the Web site from page 0/2/0 to page 0/2/0/1. However it was not considered, in this case, because the crawler has found the link from 0/2 to 0/2/0/1 first. Such "shortcuts" are common in some Web sites and we will explain how we address these anomalies later.

4.3 Navigation Pattern Generation

The goal of this phase is to build a *Navigation Pattern (NP)* from the TPM generated in the mapping phase. As we have already mentioned, the NP consists of a list of regular expressions that represent the links in a page the crawler has to follow to reach the target pages. In order to generate it, we have to generalize the URLs of the pages in the paths of the TPM using regular expressions, and select among all possible paths from the entry points the "best" one that leads to desired target pages.

The first step in this phase consists of grouping the nodes in the TPM tree that represent pages with URL that are considered *similar*. We will postpone the discussion on the notion of URL similarity for the moment. The procedure for performing this grouping is presented in Figure 4.

The **Group** procedure is initially applied to the root of the TPM tree and then recursively applied to its child nodes. In this procedure, $\mu(x)$ denotes the URL of node x, and $\mathcal{C}(x)$ denotes the set of child nodes of node x. For a given node r, Lines 3–6 define a partitioning on the set of child nodes of r, where each partition G_i contains only nodes corresponding to pages with a similar URL. To evaluate the similarity between two URLs we use the **URLsim** function (Line 6). This function will be detailed later.

```
1  Group(r)
2  begin
3    let G = {G1, G2, ..., Gk}, k ≤ |C(r)| such that:
4         G1 ∪ ... ∪ Gk = C(r)
5         Gi ∩ Gj = ∅
6         cℓ ∈ Gi iff cm ∈ Gi and URLsim(μ(cℓ), μ(cm))
7    foreach Gi = {ci1 ... cik} do
8      π = URLpattern(μ(ci1) ... μ(cik))
9      ni ← ⟨π, ci1 ∪ ... ∪ cik⟩
10     Remove all cij ∈ Gi as a child of r
11     Add ni as a child of r
12     call Group(ni)
13   end
14 end
```

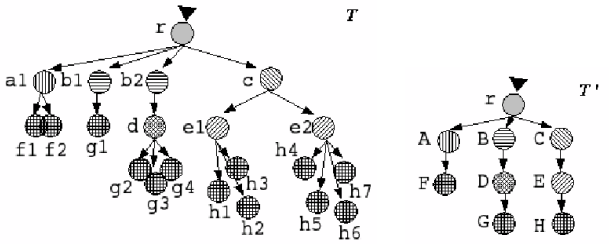
Figure 4: Grouping nodes procedure.

Exp. Id	Regular Expression	Applied in
E_1	`http://www.ejazz.com.br/artistas/default_^[a-zA-Z]+$`	Entry Point
E_2	`http://www.ejazz.com.br/detalhes-artistas.asp?cd=^d+$`	Artist Hub Page

Table 1: Example of a simple navigation pattern. A Perl-like syntax is used.

Next, in Line 11, a new node n_i is created from the nodes in G_i as follows: the URL in n_i corresponds to a regular expression π that generalizes the set of URL from the nodes in G_i according to function **URLpattern**. This function will be detailed later. The set of children of n_i is composed by all children of the nodes in G_i. Then, all the nodes in G_i are removed from the set of r children (Line 10), and replaced by the node n_i (Line 11). In summary, we replace all nodes corresponding to pages with similar URLs by a single node that represents all of them, so that there will be such a node replacing each partition of similar children. Finally, in Line 12, the **Group** procedure is recursively called for the new node n_i.

The execution of the **Group** procedure is illustrated in Figure 5, where the left side (a), illustrates a TPM T and the rigth side (b) illustrates the tree T' generated as the result of applying this procedure. Consider that nodes with the same fill pattern correspond to pages with similar URLs. Each node represented by a capital letter (`A, B, C, ...`) in T', except for the root `r`, groups all nodes from T represented by non capital letters (`a1, b2, b2, ...`). Notice that nodes labeled `A`, `C` and `D` actually represent single nodes, `a`, `c` and `d`, respectively.



(a) (b)

Figure 5: Illustration of the execution of the Group procedure.

After the execution of the grouping procedure we will possibly have more than one path leading to the target pages. This is depicted in Figure 5 where three possible paths are found: `r` to `F`, `r` to `G` and `r` to `H`. From theses paths, we choose the one that leads to the greatest number of target pages. In our example, we see that this condition is satisfied by path `r` to `H`. Thus, this path will finally be elected as the navigation pattern NP. This notion is formalized in Definition 2.

Definition 2. *Let S be a Web site and let T be a TPM for S obtained when $e, p \in S$ are respectively the entry point and the sample page supplied by the user. A NP is the path in the tree T' obtained after applying the* **Group** *procedure over T, which begins in e and ends in the node of T' that corresponds to the greatest number of target pages in T*

So far, we have postponed the discussion on how we measure the similarity between two URLs using function **URLsim** (Line 6) and on how we generate a pattern for representing a set of similar URL using function **URLpattern** (Line 8). Now we detail these two functions.

4.3.1 Evaluating Similarity Between URLs

We consider that a URL is a string formed by several substrings separated by a "/". We call these substrings *levels*. Thus, let $u = u_1/u_2/\ldots/u_n$ $(n \geq 1)$ and $v = v_1/v_2/\ldots/v_m$ $(m \geq 1)$ be two URLs. URLs u and v are considered similar if: (1) they have the same number of levels, i.e., $n = m$; (2) the first level in u and v are equal, i.e., $u_1 = v_1$; and (3) there are at most K levels in the same position in u and v (except for the first) that are not equal, i.e., let $\delta = \{\langle u_i, v_i\rangle \mid u_i \neq v_i\}$, then $|\delta| \leq K$. By performing preliminary experiments with several real URLs, we have found that a value of $K = 2$ results in an accurate and safe similarity evaluation. This result was confirmed by our final experiments.

Thus, the function **URLsim** called in Line 6 of algorithm of Figure 4 is true for two given URL u and v if they satisfy the three conditions above.

4.3.2 URL Pattern Generation

We now detail the procedure used to generate a pattern that represents a set of URLs. We recall that the URLs are assumed to be similar in the same sense as described above and that these URLs differ at most in K levels. Thus, our strategy for generating URL patterns consists in building regular expressions that match all strings occurring in each level where the URLs differ from each other. In order to do this we introduce the notion of a *Regex tree*.

Definition 3. *A* Regex tree *R is a hierarchy (a tree) in which each node n is associated with a regular expression e_n over an alphabet Σ that denotes a language L_n, such that for each internal node a in R whose children are $\{c_1, \ldots, c_k\}$ we have $L_{c_1} \cup \ldots \cup L_{c_k} = L_a$ and $L_{c_1} \cap \ldots \cap L_{c_k} = \emptyset$.*

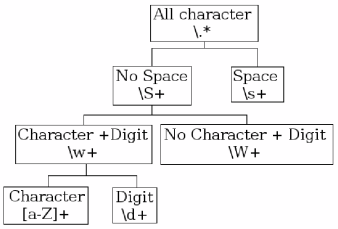


Figure 6: A Regex tree. A Perl-like syntax is used.

The Regex tree we adopt in our work is presented in Figure 6. It is an adaptation of the so called *Delimiters Tree* defined in [11]. According to Definition 3, the Regex tree is built in such a way that each node defines a language that is disjoint with the languages in its sibling nodes. In particular, each leaf node defines a language that is disjoint with the languages in its sibling leaf nodes. Thus, when applied to the tokens in a URL, a given token will match exactly one leaf node. This property allows us to use the leaf nodes in the Regex tree to tokenize a URL or a URL level.

Furthermore, each internal node defines a language that is formed by the union of the languages of its child nodes. This property allows us to find a single regular expression that matches all tokens in a same position occurring in a

set of URL. Let l_t and l_s be two leaf nodes matched by two distinct tokens t and s. The node a that is the deepest common ancestor of l_t and l_s defines a regular expression that matches t and s.

The complete procedure for generating an URL pattern is described in Figure 7, where the symbol "•" is used to denote the string concatenation operation. To explain this procedure, we will use the set of sample URLs presented in Figure 8. When taking as input these URLs, the **URLPattern** procedure iterates in the loop of Lines 7–18 over the levels in the URLs that are distinct. As illustrated in Figure 8(a), only levels 6 and 7 will be processed in this loop. We will focus on level 7, since it is more illustrative of the procedure functionality. This level is detailed in Figure 8(b). According to Line 8 there is no common prefix between all $u_i[7]$, but there is common suffix "`.html`" between them. In the loop of Lines 9–11, we tokenize each infix $f_i[7]$ generating the tokens present in Figure 8(b). Next, in the loop of Lines 12–16 each set of tokens in the same position of the infixes is given as an argument for calling the **TokenRegex** function. This means that in Figure 8(b), each column labeled $\hat{f}_i[7][j]$ will be given as an argument of **TokenRegex**. The result of each calling appears in the line labeled π. For column $\hat{f}_i[7][1]$ all tokens will match the same leaf in the Regex tree of Figure 6. Thus, the regular expression in this node is used. For column $\hat{f}_i[7][2]$, as the same toke repeats for all lines, it is simply considered as the regular expression. In the case of the tokens in column $\hat{f}_i[7][5]$, notice that the tokens "8" and "D" match distinct leafs in the Regex tree and the deepest common ancestor corresponds to the node whose regular expression is"`\w`". Thus, this regular expression is used.

4.4 Structure-Driven Crawler

In the previous section we have described how to create a navigation pattern (NP) to reach target pages in a Web site. In this section we will explain how to use the NP generated to drive a crawler by means of the procedure presented in Figure 9. Basically, it consists of a breath-first traversal of the target Web site where the links to be followed are selected according to the regular expressions in the NP.

```
1  Structure-Driven Crawler
2  input:An entry point e of a Web site S
3         A sample page p
4         A navigation pattern NP
5  output: The set P of pages structurally similar to p
6  begin
7    let Q be a queue
8    let NP = ⟨π1, ..., πn⟩
9    foreach link l from e that match π1 do
10     Q ← Q ∪ {⟨l, 2⟩}
11   end
12   while Q ≠ ∅ do
13     ⟨u, j⟩ ← Dequeue(Q)
14     g ← Download(u)
15     if j ≠ n then
16       foreach link l from g that match πj do
17         Q ← Q ∪ {⟨l, j + 1⟩}
18     else P ← P ∪ {g}
19   end
```

Figure 9: Structure-driven crawler

The algorithm uses a queue Q whose elements are pairs $\langle u, j\rangle$, where u is an URL and j indicates the current level being traversed in the Web site. In the loop of Lines 9–11 the links from the entry page that match the first regular expression π_1 in the navigation pattern are enqueued. From this point on, and while the queue is not empty, the pages in the site are visited in a breadth-first way. If the condition in Line 15 fails, it means that a target page in the last level (n) was reached. Thus, in Line 18, this page is stored. At the end, all target pages from the Web site are fetched.

5. EXPERIMENTAL RESULTS

In this section we present the results of an experimental evaluation we carried out with the proposed structure-driven crawling approach. For this we used 11 representative real Web sites which present collections of data-rich Web pages characterized by having a common structure. We have only used sites with a large number of pages and an expressive amount of target pages to be fetched. The list of the sites used, along with brief description and the definition of the target pages we wanted to fetch, is presented in Table 2.

For each site, our experiments consisted in first generating a navigation pattern using a single sample page. Next, the structure-driven crawler using this navigation pattern was executed over the site. The results are presented in Table 3.

In Table 3, the "Manual" column shows the number of pages found in each site as being target pages. These numbers were obtained by exhaustive manual navigation through the portions of each site where the target pages were located. The "Automatic" column shows the number of target pages automatically identified and fetched by the generated crawler. The recall percentage is also shown in this column. It is worth noticing that, in all cases, we reached 100% precision, since all pages fetched were manually correctly verified as target pages. The columns "Generation" and "Crawling" show the number of links traversed, respectively, for generating the crawler during the site mapping phase and for fetching the target pages during the crawling. Notice that, in all cases, the numbers in the "Crawling" column are smaller than the numbers in the "Generation" column. This occurs because, during crawling, only the links matching the regular expression in the navigation pattern are traversed.

We run each generated crawler over the corresponding Web site of Table 2 two more times. For some of them, it was detected that new pages were added after the generation of the crawler. Table 4 shows the results of these two additional crawling processes over each site. We notice that the recall figures in theO "Automatic" column remain similar to the ones in Table 3. The same can be said with respect to the number of links traversed during the crawls, which are presented in the "Traversed"column. The column "New Pages" accounts for the new target pages found by the crawler. This will occur to all new pages as long as they retain the same structure of the sample page supplied for the crawler generation. Notice that if the structure changes, a new crawler can be easily generated.

6. CONCLUSION AND FUTURE WORK

We have proposed and evaluated a new approach for automatically generating structure-driven Web crawlers. The new approach uses the structure of Web pages instead of their content to determine which pages should be collected or not. Our experiments indicate that the new structure-driven approach can be extremely effective, resulting in high precision and recall levels when crawling specific domain pages in data-rich Web sites. The proposed method also has the advantage of requiring only a few examples to learn how to identify the set of Web pages that should be fetched.

```
1  URLPattern
2  input: U = {u_1, u_2, ..., u_n}, a set of URLs
3  output: u_π, a URL pattern
4  begin
5    u_π ← u_1
6    let u_i[k] be the k−th level of URL u_i
7    foreach k such that {⟨u_1[k], ..., u_n[k]⟩ | u_i[k] ≠ u_j[k] for some 1 ≤ i, j ≤ n} do
8      let u_i[k] = p • f_i[k] • s, where p (s) is the common prefix (suffix) between all u_i[k]
9      for i ← 1 to n do
10       ⟨f_i[k][1], f_i[k][2], ..., f_i[k][m_{i,k}]⟩ ← Tokenize(f_i[k])
11     end
12     for j ← 1 to MAX {m_{1,k}, m_{2,k}, ..., m_{n,k}} do
13       x ← TokenRegex(f̂_1[k][j], f̂_2[k][j], ..., f̂_n[k][j])
14         where f̂_i[k][j] = f_1[k][j] if j ≤ m_{k,i} or tf_1[k][j] = λ otherwise
15       π ← π • x
16     end
17     u_π[k] ← s • π • p
18   end
19 end
```

```
1  Tokenize
2  input: s, a string
3  output: T = {s_1, s_2, ..., s_n}, a set of tokens
4  begin
5    let R be a Regex tree
6    i ← 0
7    while s ≠ "" do
8      i++;
9      s_i ← the leftmost largest substring
10       from s that matches a leaf in R
12     s ← s − s_i
13   end
14 end
```

```
1  TokenRegex
2  input: T = {t_1, t_2, ..., t_n}, a set of tokens
3  output: p_T, a regex matching the tokens in T
5  begin
6    if t_i = t_j for all t_i, t_j ≠ λ
7      then p_T ← t_i
8      else
9        let R be a Regex tree
10       let η(t_i) the leaf node that matches token t_i ≠ λ
11       a ← the deepest an ancestor of all η(t_i)
12       p_T ← e_a
15   if there is some t_i = λ then p_T ← p_t • *
17 end
```

Figure 7: Procedure for generating an URL pattern

Site	Description	Target Pages
www.ejazz.com.br	Pages on jazz genre, artists, instruments, etc.	Artists
informatik.uni-trier.de/∼ley/db/conf/vldb	VLDB Conference section of DBLP	VLDB conferences
www.olympic.org	International Olympic Committee Web site	Olympic heroes
www1.folha.uol.com.br/folha/turismo	Traveling section from "Folha de São Paulo" newspaper	News
www1.folha.uol.com.br/folha/dinheiro	Money section from "Folha de São Paulo" newspaper	News
dot.kde.org	Package release section from the KDE Desktop Manager	Package announcements
www.amazon.com	Amazon Essential CDs section	CD
www.wallstreetandtech.com	WallStreet and Technology News	Last news
www.cnn.com/weather	Weather in several cities around the world	Weather forecast
sports.yahoo.com	Sports section from Yahoo	European soccer teams
www.nasa.gov/home	Nasa official site	News from Nasa

Table 2: List of the Web sites used in the experiments.

Site	Target Pages		Links Traversed	
	Manual	Automatic	Generation	Crawling
E-jazz	149	149(100%)	2213	199
VLDB	30	30(100%)	70	32
OLYMPIC	335	328(98%)	395	379
Traveling	301	301(100%)	348	335
Money	470	468(99%)	550	528
KDE	30	30(100%)	120	31
CDs	416	398(96%)	440	426
WallStreet	261	253(97%)	1579	283
CNN	51	49(96%)	485	65
Yahoo Sports	38	37(97%)	1307	45
NASA	339	325(95%)	687	389

Table 3: Results of the experiments with each generated crawler.

Site	Crawl	Pages Fetched		Links	New Pages
		Manual	Automatic		
Travel	1	314	308(98%)	335	7
	2	303	291(96%)	310	11
Money	1	486	478(98%)	497	84
	2	482	476(99%)	497	80
KDE	1	29	29(100%)	31	14
	2	29	29(100%)	34	19
CDs	1	409	394(96%)	492	4
	2	418	412(98%)	487	18
WallStreet	1	267	257(96%)	271	17
	2	272	267(98%)	273	25
NASA	1	334	320(95%)	339	12
	2	337	323(95%)	341	13

Table 4: Results of crawling after the adding of new target pages.

In fact, we have used just one example in our experiments, while in the content-driven approach it is usually necessary to provide a significant set of examples in the learning phase, usually a few dozen [5, 14, 16].

The structure-driven approach is complementary to the traditional content-driven approach, in the sense that it is more suited for sites that are data intensive and, at the same time, present regular structure. This means that our new method is the best option for a restricted set of crawling tasks. However, it is important to notice that this kind of data intensive Web sites is becoming more popular as the Web grows.

As future work we plan to increase the expressiveness of the navigation patterns by adding the capability of dealing with sites in the so-called Hidden Web, i.e, those sites whose

(a)

	$u_i[1]$	$u_i[2]$	$u_i[3]$	$u_i[4]$	$u_i[5]$	$u_i[6]$	$u_i[7]$
u_1	www.informatik.uni-trier.de	∼ley	db	indices	a-tree	c	Chaves:Silvio_8=.html
u_2	www.informatik.uni-trier.de	∼ley	db	indices	a-tree	u	Ugher:Mangabeira_D=.html
u_3	www.informatik.uni-trier.de	∼ley	db	indices	a-tree	m	Mendes:Sergio.html
Π	www.informatik.uni-trier.de	∼ley	db	indices	a-tree	[a-Z]+	[a-Z]+:[a-Z]+_*\w*.html

(b)

	$\hat{f}_i[7][1]$	$\hat{f}_i[7][2]$	$\hat{f}_i[7][3]$	$\hat{f}_i[7][4]$	$\hat{f}_i[7][5]$	s
$u_1[7]$	Chaves	:	Silvio	_	8	.html
$u_2[7]$	Ugher	:	Mangabeira	_	D	.html
$u_3[7]$	Mendes	:	Sergio	λ	λ	.html
π	$[a-Z]+$	:	$[a-Z]+$	_*	\w*	.html

Figure 8: Example of a simple navigation pattern. A Perl-like syntax is used.

pages are automatically generated from results of queries to databases using arguments taken from forms. Another future extension we envision is the combination of the content-driven and structure-driven approaches, creating a hybrid strategy that combines the advantages of using content and structure. The idea is to produce methods that are more flexible than the structure-driven method proposed here, while still achieving high precision and recall levels.

7. ACKNOLEDGEMENTS

This work is partially supported by projects GERINDO (CNPq/CT-INFO 552.087/02-5) and SIRIAA (CNPq/CT-Amazônia 55.3126/2005-9) and individual grants by CNPq (303032/2004-9 Altigran S. da Silva) (303576/2004-9 Edleno S. de Moura) and FAPEAM (Marcio Vidal). This research is also sponsored by UOL (www.uol.com.br), through its "UOL Bolsa Pesquisa" program, Proc. num. 200503301456.

8. REFERENCES

[1] Aggarwal, C. C., Al-Garawi, F., and Yu, P. S. On the design of a learning crawler for topical resource discovery. *ACM Transactions on Information Systems 19*, 3 (2001), 286–309.

[2] Ahnizeret, K., et al. Information retrieval aware web site modelling and generation. In *Proceedings of the 23rd International Conference on Conceptual Modeling* (Shanghai, China, 2004), pp. 402–419.

[3] Arasu, A., and Garcia-Molina, H. Extracting structured data from web pages. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data* (San Diego, CA, USA, 2003), pp. 337–348.

[4] Calado, P., et al. Web-dl: an experience in building digital libraries from the web. In *Proceedings of the ACM International Conference on Information and Knowledge Management* (McLean, VA, USA, 2002), pp. 675–677.

[5] Chakrabarti, S., Punera, K., and Subramanyam, M. Accelerated focused crawling through online relevance feedback. In *Proceedings of the 11th International World Wide Web Conference* (Honolulu, HI, USA, 2002), pp. 148–159.

[6] Chakrabarti, S., van den Berg, M., and Dom, B. Focused crawling: A new approach to topic-specific web resource discovery. *Computer Networks 31*, 11-16 (1999), 1623–1640.

[7] Chawathe, S. S. Comparing hierarchical data in external memory. In *Proceedings of the Twenty-fifth International Conference on Very Large Data Bases* (Edinburgh, Scotland, U.K., 1999), pp. 90–101.

[8] Cooley, R. The use of web structure and content to identify subjectively interesting web usage patterns. *ACM Transactions on Internet Technology 3*, 2 (2003), 93–116.

[9] Crescenzi, V., Merialdo, P., and Missier, P. Clustering web pages based on their structure. *Data and Knowledge Engineering 54*, 3 (2004), 277–393.

[10] Davulcu, H., et al. A layered architecture for querying dynamic web content. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Philadelphia, PY, USA, 1999), pp. 491–502.

[11] de Castro Reis, D., et al. A framework for generating attribute extractors for web data sources. In *Proceedings of the 9th International Symposium on String Processing and Information Retrieval* (Lisbon, Portugal, 2002), pp. 210–226.

[12] de Castro Reis, D., et al. Automatic web news extraction using tree edit distance. In *Proceedings of the 13th international conference on World Wide Web* (New York, NY, USA, 2004), pp. 502–511.

[13] Lage, J. P., et al. Automatic generation of agents for collecting hidden web pages for data extraction. *Data an Knowledge Engineering 49*, 2 (2004), 177–196.

[14] Liu, H., Milios, E. E., and Janssen, J. Probabilistic models for focused web crawling. In *Procendings of the ACM CIKM International Workshop on Web Information and Data Management* (Washington, DC, USA, 2004), pp. 16–22.

[15] Najork, M., and Wiener, J. L. Breadth-first crawling yields high-quality pages. In *Proceedings of the 10th International World Wide Web Conference* (Hong Kong, China, 2001), pp. 114–118.

[16] Qin, J., Zhou, Y., and Chau, M. Building domain-specific web collections for scientific digital libraries: a meta-search enhanced focused crawling method. In *Joint Conference on Digital Libraries* (Tuscon, AZ, USA, 2004), pp. 135–141.

[17] Selkow, S. M. The tree-to-tree editing problem. *Information Processing Letters 6* (Dec. 1977), 184–186.

[18] Yang, W. Identifying syntactic differences between two programs. *Software – Practice And Experience 21*, 7 (July 1991), 739–755.

[19] Zhai, Y., and Liu, B. Web data extraction based on partial tree alignment. In *Proceedings of the 14th international conference on World Wide Web* (Chiba, Japan, 2005), pp. 76–85.