A Novel Web Archiving Approach based on Visual Pages Analysis

Myriam Ben Saad
LIP6, University P. and M. Curie
Paris, France
Myriam.Ben-Saad@lip6.fr

Stéphane Gançarski
LIP6, University P. and M. Curie
Paris, France
Stephane.Gancarski@lip6.fr

Zeynep Pehlivan
LIP6, University P. and M. Curie
Paris, France
Zeynep.Pehlivan@lip6.fr

ABSTRACT

Due to the growing importance of the World Wide Web, archiving the web has become a cultural necessity in preserving knowledge. To maintain a web archive up-to-date, crawlers harvest the web by iteratively downloading new versions of documents. However, it is frequent that crawlers retrieve pages with unimportant changes such as advertisements which are continually updated. Hence, web archive systems waste time and space for indexing and storing useless page versions. In this paper, we present a novel approach that detects important changes between versions in order to efficiently archive the web. Our approach combines the concept of the visual pages segmentation with the concept of importance while detecting changes between versions. The approach consists of archiving the visual layout structure of a web page represented by semantic blocks. We propose an adequate changes detection algorithm to compute differences between these visual layout structures of documents. We describe also a method to evaluate the importance of detected changes. Tests were conducted to evaluate the feasibility of our approach. Experimental results show promising performances of our approach.

Keywords

Web archiving, change detection, visual pages analysis

1. MOTIVATION

With the explosive growth of the Internet, the web has become the largest repository of content (on-line newspapers, articles, published documents, etc.). Such content provides a big amount of useful knowledge in many areas. Archiving the web is crucial for preserving useful source of information. For this reason, it has become an issue for many national archiving institutes around the world. However, the web is highly dynamic, evolving over time (pages change frequently). Most often, web archiving is automatically performed using web crawlers. Web crawlers visit web pages to be archived and build a snapshot and/or index of web pages. In order to maintain the archive up-to-date, crawler must revisit periodically the pages and update the archive with fresh images. However, the crawler can not revisit a site and download a new version of a page all the time because it usually has limited resources (such as bandwidth, space storage, etc.) with respect to the huge amount of pages to archive. In fact, it is impossible to maintain a complete archive of the whole Web, or even a part of it, containing all the versions of all the pages. Thus, the problem can be stated as follows: how to optimize the crawling in order to download the "most important versions", so that the minimum of useful information is lost? Of course, this problem must be solved without any help from web sites managers. Thus, the archiving system must estimate the behavior of a site in order to guess when or with which frequency it must be visited.

Several works [13, 12, 27] have focused on estimating the change frequency to improve the Web crawlers. However, the crawler may waste time and space to download a new page version with unimportant changes such as advertisements which are continually updated. Thus, an effective method is required to know accurately when and how often important changes between versions occur. Up to now, approaches that estimate the frequency of changes only take into account the amount of detected changes. But, they do not consider the *importance* of changes that have occurred. If we can predict important changes frequencies much more accurately, we may improve the effectiveness of the web archive system. Therefore, the crawler can improve the freshness of the local snapshot without consuming as much bandwidth. It also avoids indexing unimportant information and wasting space storage. The counterpart is that processing page versions to detect changes does not come for free and we must take care of its complexity so that it does not become the bottleneck of the system.

In order to estimate the frequency of updates, changes between already retrieved versions of documents must be detected. Many existing algorithms [21, 15, 29] have been specially designed to detect changes between semi-structured documents (XML and HTML). However, there is no method that detects and distinguishes relevant/irrelevant changes from useful/useless (noisy) information.

Our work[1] proposes an approach for detecting important changes between versions in order to efficiently archive the web. It will be applied on a repository for the French Na-

[1]This research was supported by the French National Research Agency ANR in the CARTEC Project (ANR-07-MDCO-016).

tional Audiovisual Institute (INA). One of the missions of INA is to create a legal deposit which preserves French radios and televisions web pages and related pages. A strong requirement for this project is that the visual aspect of the pages must be preserved. Thus our idea is to use a visual page analysis to assign importance to web pages parts, according to their relative location. In other words, page versions are restructured according to their visual representation. Detecting changes on such restructured page versions gives relevant information for understanding the dynamics of the web sites.

Previous works [8, 17] show that a page can be partitioned into multiple segments or blocks and, often, the blocks in a page have a different importance. In fact, different regions inside a web page have different importance weights according to their location, area size, content, etc. Typically, the most important information is on the center of a page, advertisement is on the header or on the left side and copyright is on the footer. Once the page is segmented, then a relative importance must be assigned to each block. This can be achieved automatically using for instance the algorithm of [28], or though a supervised machine learning method. Then, we can compute the importance of changes between two page versions, based on (1) the relative importance of blocks and (2) the relative importance of operations (insert, delete, update, etc.) occurred in those blocks, detected by comparing the two versions.

The concepts of visual page analysis and importance of web pages parts are not new. However, as far as we know, they had never been combined for archiving the web. Our focus, in this paper, is on detecting important web pages changes and optimizing the use of allocated resources. Our main contributions can be summarized as follows:

- A novel web archiving approach that combines three concepts: visual page analysis (or segmentation), visual change detection and importance of web page's blocks.

- An extension of an existing visual segmentation model to describe the whole visual aspect of the web page.

- An adequate change detection algorithm that computes changes between visual layout structures of web pages with a reasonable complexity in time.

- A method to evaluate the importance of changes occurred between consecutive versions of documents.

- An implementation of our approach and some experiments to demonstrate its feasibility.

This paper is structured as follows. In Section 2, related works are discussed. Section 3 presents the architecture of the web archive system and some useful concepts. Section 4 describes the extended visual page segmentation model for HTML web pages. In Section 5, we propose an adequate change detection algorithm to compute differences between two visually restructured page versions. Section 6 describes the method for evaluating the importance of blocks/changes. Section 7 presents the implementation and discusses experimental results. Section 8 concludes.

2. RELATED WORK

As mentioned in the introduction, our work is related to web archiving, visual web page analysis and change detection. We present below related works in those three areas.

Web Archiving. There are several projects launched by different archiving institutes (national libraries, historical data archives, etc.) around the world to preserve their country's web heritage. These institutes intend to provide future generation with relevant past (versions of) web pages. Most of the web archiving initiatives are described at [3]. Archiving is made either manually (Australia [2], Canada [24]) or by an automatic process based on crawlers (Nordic countries [6], the Internet Archive [1], French National Libraries [4]). Some studies [4, 7] focus on the selection of web pages to be archived by defining the web perimeter. Others [13, 12, 27] work on modeling and evaluating the frequency of web changes. They propose change frequencies estimators and various refresh policies to improve the archive freshness. Some researchers [9, 18] address issues concerning the format of information to be stored and indexed by proposing their own storage system. Others studies [4, 5] focus on the control and the representation of changes. They propose a change detection algorithm and/or a delta format for an efficient query and storage of the web archive.

Though interesting, those approaches do not take the visual aspect and the relative importance of pages parts, which are at the core of our approach.

Visual Web Page Analysis. Several methods have been proposed to analyze the visual representation of web pages. The visual aspect of the document gives a good idea of the semantic structure used in the document and the relations among them. Most approaches discover the logical structure of a page by either analyzing the rendered document or analyzing the document code. Yang and Zhang [30] describe an approach to extract and analyze the semantic structure of HTML documents derived directly from the page layout. Their approach tries to detect visual similarities between HTML content objects. Gu et al. [19] propose a top down algorithm based on the layout information. They detect the web content structure by dividing and merging blocks. Kovacevic et al. [17] build up a M-tree that represents the structure of the page. Then, based on visual information of each M-tree's node, they define heuristics to recognize common page areas (header, left and right menus, footer and center of the page). Cai et al. [8] propose a top down, tag tree independent approach that extracts the web content structure based on visual information given by the web browser. Their VIPS algorithm segments the web page into multiple semantic blocks based on visual information retrieved from browser's rendering. Cosulshi et al. [16] propose an approach that calculates the block correspondence between web pages by using positional information of DOM tree's elements. In our project, we are interested by archiving the visual aspect of radio and television web pages. Therefore, we intend to detect changes between versions based on the visual representation of web pages. In this context, the VIPS method described above seems to be the most appropriate because it allows an adequate granularity of the page partitioning. It extracts suitable blocks from the HTML DOM tree, based on visual information retrieved from browser's rendering. Compared to existing methods, VIPS builds a hierarchy of semantic blocks of the page that better simulates how a user understands the web layout structure based on his visual perception. We extend the VIPS model to generate, as output, a *Vi-XML* document describing the whole visual structure of the web page. This Vi-XML document is used for detecting changes. The choice of using XML as language is motivated by the need to ease exchanging, comparing, storing and querying the different versions of archived pages.

Change Detection. Several algorithms have been designed to detect changes between two (versions of) documents. Previous works in change detection have dealt with flat files but our focus here is on hierarchical data such as XML documents. The hierarchical change detection problem is defined as tree-edit-distance. It consists in finding a minimum set of change operations (insert, delete, ...) that transform one data tree to another. These changes operations are often gathered in a delta script or a delta file.

Various algorithms have been proposed for finding changes between XML documents. These algorithms have different complexities, memory usages and delta formats. The complexity mainly depends on the data manipulation operations that the algorithm is able to detect. Some algorithms are fast but they do not produce an optimal delta script and/or do not detect move operation (instead, they detect a delete/insert couple which is semantically poorer). Others handle a move or a copy in addition to basic operations that can impact positively in the delta size. However, this can increase the complexity time of algorithms. The design of those diff algorithms depends on the purposes and the requirements (time complexity, operations to be handled, quality of the delta, etc.).

Chawathe proposes an algorithm LaDiff [11] which supports move operation for ordered tree in addition to basic operations (insert, delete and update). It achieves at time complexity of $O(n * e + e^2)$ where n is the total number of leaf nodes and e is the weighted edit distance between two trees. Chawathe also proposes MH-Diff [10] for unordered trees with move and copy operations, with a time complexity of $O(n^2 * log(n))$. Cobéna et al. [15] propose the Xy-Diff algorithm to improve time and memory management. XyDiff also supports move operation and achieves a time complexity of $O(n * log(n))$. Despite its high performance, it does not always guarantee an optimal result (*i.e.* minimal edit script). Wang et al. [29] propose X-Diff which can detect the optimal differences between two unordered XML trees in quadratic time $O(n^2)$ but it does not handle a move. DeltaXML [21] proposed by Robin La Fontaine is the market leader. It can compare, merge and synchronize XML documents for ordered and unordered trees by supporting basic operations but it does not detect a move. Although it runs extremely fast and its results are close to minimum, it is limited in the size of the trees it can handle (maximum tree size is 50 MB). There are several other algorithms like Fast XML DIFF [23], DTD-Diff [22], etc. Luuk Peters [26] and Grégory Cobéna [14] present a survey of some algorithms and provide a comparison between them. In our project, we aim to detect changes between two versions of pages, *i.e.* between two Vi-XML documents. After studying these algorithms, we decide to not use existing change detection methods for our novel web archiving approach because they are generic-purpose. As we have various specific requirements related to the visual layout structure of documents, we prefer proposing an *ad'hoc* algorithm that will be more adequate to the visual aspect of the page. The output delta of our new algorithm is formatted according to the specific block structure of documents. Moreover, it allows for a better trade-off between complexity and completeness of the detected operations set.

3. SYSTEM MODEL

In this section, an overview of the web archive architecture is given. Then some concepts that will be used through this paper are defined.

3.1 Web Archive Architecture

Our system consists of four major components: the *web crawlers*, the *freshness component*, the *storage component* and the *query engine*. Figure 1 presents an overview of the system.

The Web Crawler. The web crawlers harvest the web by

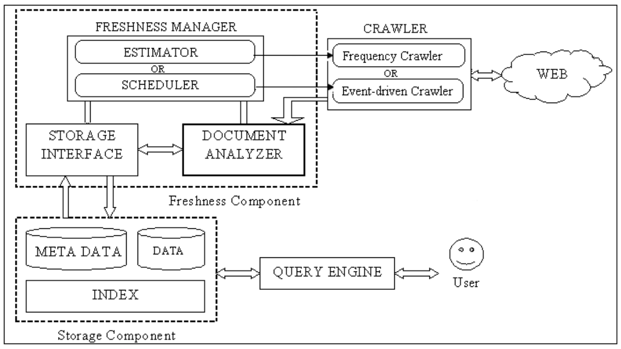


Figure 1: An Overview of the Web Archive.

iteratively downloading documents referenced by URLs. We consider two types of web crawlers. The first one downloads automatically web pages according to a certain frequency given by an estimator. The second one is an event-driven crawler. It downloads the most urgent document to be refreshed, as computed by a scheduler.

The Freshness Component. The freshness component allows maintaining the archive up-to-date. It consists of three main modules: The *freshness manager* enables the optimization of allocated resources, so that less information is lost. It consists of either a *change frequency estimator* or a *scheduler*. The estimator computes the best change frequency for the first type of crawlers. The scheduler chooses the most urgent page to be downloaded by the event driven crawler in order to maintain the archive as up-to-date as possible. It manages a list of documents ordered by a freshness urgency function. This function estimates, for each page, how it is urgent to refresh it at a given date. Both estimator and scheduler depend on the changes already detected and quantified by the document analyzer on previously archived versions. They take also into account the estimation of changes importance occurred between successive downloaded versions. Based on the freshness urgency function, the scheduler can organize and order the list of pages to be urgently refreshed.

The *Document Analyzer* builds the visual page layout structure. Then, it detects changes and quantifies their importance in order to estimate either the frequency of web crawlers or the freshness urgency function. This module is described in more details in the next paragraph. Then, page versions are stored in a database through the *storage interface* with additional information such as URL, date/time of crawl, etc. Changes are also stored to enable querying the archive with predicates about updates occurred between versions. The storage interface interacts with the storage component to store/index page versions and their metadata obtained during the analysis.

The Storage Component. The storage component consists of data and metadata storage units. It includes also an index that facilitates querying the archive.

The Query Engine. Users can navigate temporally between versions and request archived page versions through the query engine.

The document analyzer. We give here more details on the document analyzer since it is the core of our approach. It consists of several sub-modules corresponding to the various phases of the page analysis. It is depicted in Figure 2. As mentioned in Section 2, we choose to analyze changes between versions based on the visual representation of web pages. The visual web pages analysis enables to obtain a better partition of a web page at semantic level that can help to better evaluate of the importance of detected changes. In other words, comparing two pages based on their visual representation is semantically more informative than with their HTML representation.

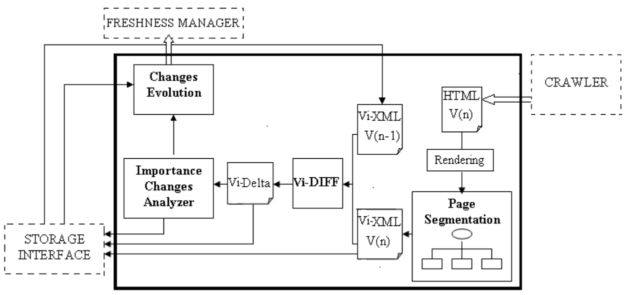


Figure 2: The Document Analyzer.

The document analyzer interacts with the crawler to get the current version of the HTML page to be archived. Then, the page is treated by a rendering engine in order to retrieve visual information. The main advantage of rendering is providing a real and a complete visual description of the document even if embedded scripts, such as JavaScript, are included. This is important since scripts can affect the behavior and the structure of the page. After that, the rendered page is partitioned and the visual page layout structure is built by analyzing the rendered page. As mentioned in Section 2, the VIPS algorithm is used to segment a web page into semantic hierarchical blocks. We extended it to extract the links, images and texts for each block. The extended VIPS algorithm generates, as output, a *Vi-XML* document that describes the hierarchical content structure of the page. At the end of the segmentation process, a change detection algorithm (*Vi-DIFF*) provides a description of changes that occurred between the new generated Vi-XML version $V(n)$ and the last version archived $V(n-1)$. Changes are gathered in a delta XML file, called *Vi-Delta*, that describes the operations (insertion, deletion, etc.) occurred between two documents. Thereafter, the Vi-Delta file is analyzed by the submodule *Importance Changes Analyzer* to evaluate the importance of detected changes. The result of this change evaluation can be used by the sub module *Changes Evolution* either to improve the estimation of the crawler frequency or to compute the freshness urgency function used by scheduler. At the end, the Vi-Delta, the current Vi-XML version and additional metadata are stored in the database through the storage interface. Further study is necessary to work on the best storage strategies to be chosen.

3.2 Concepts

In order to better understand the next Sections, we define/discuss the different data formats we manipulate.

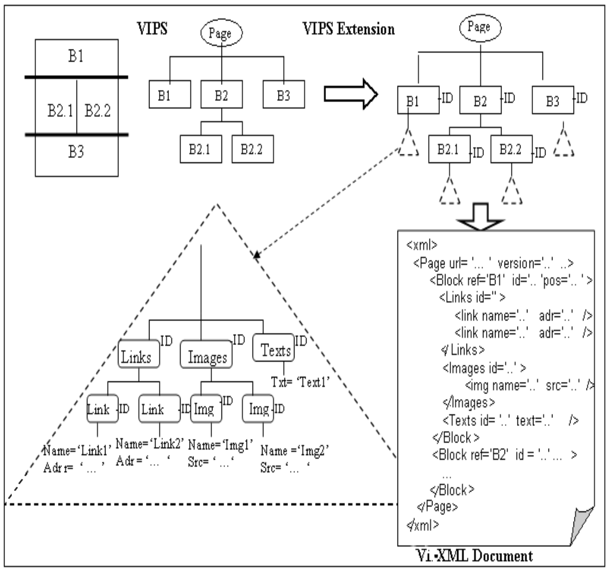
HTML The extreme simplicity of HTML has played an important role to make it popular and widely used in the web. Several languages can be used to produce HTML pages like PhP, Sun JSP, and Microsoft ASP. It can include embedded scripting language code (such as JavaScript) that can affect the behavior of web browsers. In our project, we are interested by archiving HTML web pages even if they can have eventual complex structures (JavaScript, etc.), which means that we must work on the rendered pages, not only on the HTML code. We are currently working on optimizing the rendering process, but this issue is beyond the scope of this paper.

Vi-XML is an XML document that describes the tree structure of the visual aspect of web page version. As mentioned before, this document is produced by our extended VIPS which constructs the hierarchical semantic blocks of the page. The root of Vi-XML tree is the document and has as descendants multiple (nested) blocks which represent the page's regions. Each leaf block has additional children nodes like links, images and text. The choice of using XML is motivated by the need to easily exchange, store and query the different page versions. It enables representing the visual aspect of a page version at a semantic level.

Vi-Delta is an XML document describing the sequence of change operations (insert, delete, etc.) needed to transform one Vi-XML page version to the next one. It is produced by our change detection algorithm Vi-DIFF, described in Section 5. Vi-Delta has a specific format related the visual block structure of the web page. It is composed of several elements, each node storing detected change operations of a given type : (i) element nodes (links, images, or text) deleted from a block, (ii) element nodes inserted into a block, (iii) updated element nodes, (iv) nodes moved from one block to another. Stored changes are first organized and grouped by block then by operations type. As for Vi-XML documents, we choose XML as language to represent Vi-Delta because we want to query changes detected between two versions of document. Existing query language for XML such as XQuery can then be used.

4. VISUAL PAGE SEGMENTATION

As mentioned before, VIPS [8] is used to segment a web page into nested semantic blocks based on suitable nodes in the HTML DOM tree of the page. It detects the horizontal and vertical separators in a web page. Based on those separators, it builds the semantic tree of the web page partitioned into multiple blocks. The root is the whole page. Each block is represented as a node in the tree as shown in Figure 3. To complete the semantic tree of the whole



```
<xml>
  <Page url= '... ' version='..' ..>
     <Block ref='B1' id='.. 'pos='.. ' >
       <Links id='' >
          <link name='..'  adr='..'  />
          <link name='..'  adr='..'  />
       </Links>
       <Images id='..' >
            <img name='..' src='..' />
       </Images>
       <Texts id= '..' text='..'   />
     </Block>
     <Block ref='B2' id = '..' ...  >
       ...
     </Block>
  </Page>
</xml>
```

Figure 3: The Extended VIPS Algorithm.

page, we extended the VIPS algorithm by extracting links, images and text for each block. As illustrated in Figure 3, each block node has additional children nodes: Links, Images and Texts that gather respectively all hyperlinks, pictures and text contained in the block. All nodes of the page are uniquely identified by an ID attribute. This ID is a hash value computed using the node's content and its children nodes content: if matched nodes (nodes at the same position in two successive versions) have different ID values, then their content has been necessarily updated. Leaf nodes have other attributes such as the name and the address for the hyperlink. Our extended VIPS algorithm generates, as output, a Vi-XML document that describes the complete hierarchical structure of the web page. The structure of such a document is shown in Figure 3.

5. CHANGES DETECTION

In this section, we present our changes detection algorithm called *Vi-DIFF*. It computes differences between two versions of a Vi-XML document and produces a Vi-Delta document that represents the delta between the two versions. Vi-XML and Vi-Delta documents are structured as previously described. In this paper, we do not deal with changes in the blocks structure : we assume that, from one version of the document to another, only the content of blocks is modified. This is the case, for instance, for the most radio and television web sites. We are currently studying other cases, where the structure may change between versions, but they are not considered in the following.

As presented in Section 2, different change detection algorithms for XML document have been proposed. They are generic algorithms for any document, thus they do not completely satisfy our requirements. We would like to add some specific criteria for comparing attributes nodes. For instance, we would like detecting updated links, if its attribute Address is modified. We want also to detect an update text in two matched blocks based on a textual similarity/distance score (*e.g.* number of shared words): if the score is higher than 0.5, texts are considered as updated. With generic algorithms, these nodes would be considered as deleted from the old version and added to the new one. Another specificity of our approach is that we need to detect changed elements inside a block and moved elements from one block to another, but detecting moved element inside a same block is useless because no information has been added or deleted in the block. To sum up, we want that the Vi-Delta created by the Vi-DIFF simulates how a user understand changes in a web layout structure based on his visual perception. It includes insert, delete, update and inter-block move operations, which are detailed in the next section.

5.1 Change Operations

Given two versions of a Vi-XML document, a delta is composed of elementary operations that convert one version of document into another. x denotes a link, an image or a text node, *attr* and *attr'* are attributes of x, b and b' are two block nodes. The change operations are defined as follows:

- **Insert**(x(attr),b): inserts a node x with attributes *attr* in the block b. We do not deal with the position where x is inserted in b.
- **Delete**(x(attr),b): deletes a node x from the block b.
- **Update**(x(attr),x(attr'),b): changes the attributes value *attr* of a node x in the block b to a new value *attr'*.
- **Move**(x(attr),b,b'): moves a node x from block b to block b'.

Example 1.

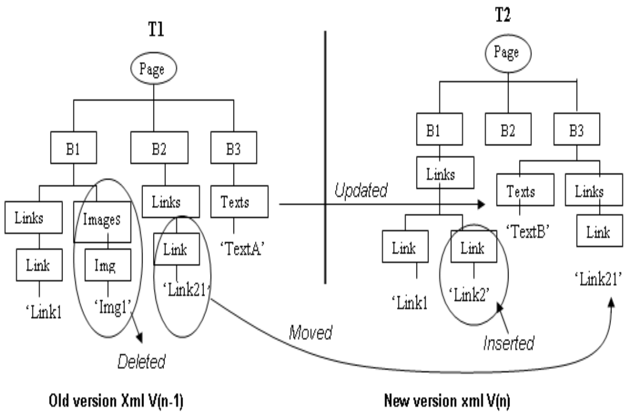


Figure 4: Change Detection Example.

Given the two Vi-XML trees T1 and T2 shown in Figure 4, the operations that transform T1 into T2 are given by the following sequence:

1. insert(link('Link2'),B1),
2. delete(img('Img1'),B1),
3. update(texts('TextA'), texts('TextB'), B3),
4. move(link('Link21'),B2,B3).

The Vi-Delta describing changes from T1 to T2 is represented in Figure 5. Detected changes are organized in the delta by block and by operations type.

```
<XML>
  <Delta from='V(n-1)' To='V(n)' >
    <Block ref='B1' ...>
      <Insert>
        <link name='Link2' adr='...' />
      </Insert>
      <Delete>
        <img name='Img1' src='...' />
      </Delete>
    </Block>
    <Block ref='B3' ...>
      <Update>
        <Texts oldText='TextA' newtext='TextB'/>
      </Update>
    </Block>
    <Move>
      <link name='Link21' adr='..' fromBlock='B2' toBlock='B3'/>
    </Move>
  </Delta>
</XML>
```

Figure 5: Vi-Delta Example.

5.2 The Vi-DIFF Algorithm

In this section, we present our Vi-DIFF algorithm that computes the differences between two Vi-XML documents and constructs a Vi-Delta to represent changes between them. In its current version, our algorithm starts with parsing two Vi-XML documents into trees. We do not consider it as a step in this paper, because the visual segmentation and the Vi-DIFF phases will be integrated together. Thus, we assume that the two Vi-XML trees are already available. Also, we do not take into account change in the visual structure. We assume that the structure does not change. The detection of changes in the structure will be treated in future works. A global overview of the Vi-DIFF algorithm is shown in Figure 6.

The steps of our Vi-DIFF algorithm can be summarized as follows:

Step 1: Initialization. Assuming the structure of blocks is fixed lets us traverse two trees (T1, T2) at the same time. T1 represents the last downloaded version of a page, T2 the one downloaded just before. First, we get all blocks in two lists (one for T1, another for T2) and then in one loop we compare block's IDs. As we mentioned in section 4, each block has an attribute ID whose value is the hash value computed using the node's content. If two blocks have the same ID, they are considered as equal in both versions. If not, the ID of each block child's (Links, Imgs, Txts) are pairwise compared. If they are changed, we add them in different arrays for each tree and each element. At the end of this step, we have two arrays for each changed element (Links, Imgs, Txts) and one empty tree for the delta. Each time we detect a change in the next step, we create a node and add it to this delta tree.

Step 2: Detecting changes. In this step, the arrays generated in step 1 are compared. As links and images have nearly the same structure, they can be represented in the same object. For texts, we need to find text similarity, thus they are treated differently.

◦ *Links and Images*:

Before starting to compare, we merge-sort each array according to the *Name* attribute. We define two counters (one

```
Input: Vi-XML Tree1, Vi-XML Tree2
Output: Vi-Delta
1. Traverse Tree1, get list of blocks B1
2. Traverse Tree2, get list of blocks B2
3. Create Delta Tree DT
4. For all block(i) in B1 do
5.    if (B1[i].ID != B2[i].ID) then
6.      Compare their Links/Img ID
7.      if Ids are different then
8.        Create object from node( ID,Name,Src,Ref )
9.        add objects of Tree1 in lists Link1/Img1
10.       add objects of Tree2 in lists Link2/Img2
11.     end if
12.   end if
13. DetectChanges (Link1, Link2, DT)
14. DetectChanges (Img1, Img2, DT)
15. DetectTextChanges (Txts1, Texts2, DT) /* find the operation
by computing the distance between two texts and add it to DT */
16. Save DT

procedure DetectChanges (List1, List2: List, DT: Delta Tree)
1. Sort both arrays with merge sort (Name attribute)
2. Define two counter variables (for each array).
3. Start to advance both counters on both arrays in an endless loop.
4.    if IDs are equal then
5.      if if Refs are different then
6.        add node for MOVE in DT
7.        advance both counters
8.      end if
9.    else if Names are equal but Adr/Src are different then
10.     add node for UPDATE in DT
11.     advance both counters
12.   else if List1.Name < List2.Name (alphabetically) then
13.     If not the end of List1 then
14.       add List1's obj for DELETE in DT
15.       advance List1's counter
16.     else the element of List2
17.       add List2's obj for INSERTED in DT
18.       advance List2's counter
19.     end if
20.   else
21.     if not the end of List2 then
22.       add List2's obj for INSERT in DT
23.       advance List2's counter
24.     else the element of List1
25.       add List1's obj for DELETE in DT
26.       advance List1's counter
27.     end if
28.   end if
29. If the end of one array then
30.   advance the counter of other one.
31. end if
32. Break at the end of two arrays
endproc
```

Figure 6: Vi-DIFF Algorithm.

for each array) and increment them in an endless loop. If ID values referenced by counters in both arrays are the same, their *Ref* (reference to the parent block) values are compared to get moved nodes: if the *Ref* values are different, it means that the mentioned node is moved from one block to another, else they are the same. Both counters are incremented. If they have different IDs, the nodes with the same Name but with the different Adr/Src. Both counters are incremented. If they are completely different, element of the first array (A1 of the first document) has a smaller value (Name attribute value ordered alphabetically), it is considered as deleted. We advance the counter of the first array. If it has a bigger value, element of the second array (A2 of the second document) is considered as inserted. Only the counter of the second array is incremented. If we reach at the end of the first array, we continue incrementing the second counter and we consider all the elements in second array as inserted. If the second array is reached to end, the

first array's counter is incremented and all its elements are considered as deleted.
○ *For Texts*:
We need to find the distance between two texts to decide if it is an update operation or a delete+insert operation. The number of different words between two texts are divided by the length of the first text. If this value is more than 0.5, the texts are considered as different texts and we have two operations delete then insert. Otherwise, the text is considered as updated.
Step 3: Save Delta Tree as XML. In this step, the delta tree, created in the first step and filled during the second step is saved as an XML file.

Complexity
We now briefly analyze the complexity of our algorithm. As explained in the introduction of this paper, this issue is important because we do not want that page processing become a bottleneck of our system. We give the complexity of worst case in which all blocks of a page changed from one version to another. The Vi-DIFF algorithm has $O(n*log(n))$ as time complexity, where n is the total number of block nodes. The complexity of the first step is $O(n)$ because we traverse each Vi-XML tree. For the second step, we use Collections.sort (in java) which has quite low complexity [25], $O(n*log(n))$. We also tried with our own implementation of merge-sort but the results were a little bit slower. For detecting changes, it consists more or less of merging two sorted arrays, so the complexity is $O(n)$. Thus, our Vi-DIFF algorithm is log-linear and achieves a time complexity of $O(n*log(n))$ which is rather promising. The main benefit we get from developing an ad'hoc algorithm is generating a specific format of Vi-Delta at reasonable time. In fact, our Vi-DIFF produces a Vi-Delta that completely satisfies our requirements related to the specific visual structure of Vi-XML files, as mentioned in Section 5. Note that in this version of the algorithm, a link/image is considered updated (and not inserted/deleted) if it has the same name but a different address. One could consider that a link/image is updated if it has the same address but a different name which would lead to symmetrical version of the algorithm, with the same complexity. Finally, one could consider that a link/image is updated if it has the same name OR the same address in both version. In this case the complexity is a little bit higher, but remains acceptable.

6. CHANGES IMPORTANCE

Based on the Vi-Delta produced by our Vi-DIFF, we now evaluate the importance of detected changes which is the task of the sub-module *Importance Changes Analyzer* (Figure 2). We aim to provide a function that enables to distinguish unimportant changes from important ones between two page versions. This function takes as input the Vi-Delta and computes a normalized value that estimates the importance of changes. This value depends on three major parameters:

• **Block Importance.** The page is segmented into multiple blocks which do not have the same importance. This importance varies according to the location, area size and content of the block. Typically, the most important information is on the center and the advertisements are on the header, etc. Song and al. [28] propose to assign automatically importance values for different blocks in the web page. Based on extracted spatial and content features, they use supervised machine learning algorithms to assign importance to different blocks. We can, also, take into account other parameters to evaluate the importance of a block with respect to the history of changes on this block. For instance, we can consider that the more frequent a block is changing, the less important it is. Further study is necessary to find the best technique to estimate the importance of blocks.

• **Operations Importance.** The importance of operations depends on the operation type (move, insert, etc.) and the changed element (link, image, etc.). For instance, insert or delete operations can be considered more important than a move. Also, inserting an image can be more important than inserting a link or a text. Again, we plan to study machine learning methods to choose the best parameters values for each operation type.

• **Changes Amount per Block.** The amount of change operations (delete, insert, etc.) occurred inside a block for each element (link, image and text) is deduced from the generated Vi-Delta. This amount represents the percentage of change operations detected for each block divided by the total number of block's elements.

Based on these parameters, we propose the following function $E(v_1, v_2)$ to estimates the importance of changes between versions v_1 and v_2, each composed of blocks Bk_i :

$$E = \sum_{i=1}^{N_{Bk}} I(Bk_i) * [\frac{1}{N_{Op}} \sum_{j=1}^{N_{Op}} I(Op_j) * \frac{1}{N_{El}} \sum_{k=1}^{N_{El}} \frac{N(Op_j, El_k)}{N(El_k, bk_i)}]$$

where:
- Op_j={insert, delete, update, move}
- El_k={link, image, text}
- N_{El} is the number of elements type in the block.
- N_{Op} is the number of operation type in the block.
- N_{Bk} is the number of block in the page.
- $I(x)$ denotes the importance value of x which can be a block or a change operation. In order to normalize the result of function $E()$, we add the following constraint on the importance of blocks : $\sum_{i=1}^{N_{Bk}} I(Bk_i) = 1; 0 \leq I(Op) \leq 1$
- $N(Op_j, El_k)$ denotes the number of change operation j that occurred on the element k.
- $N(El_k, Bk_i)$ denotes the total number of elements k inside the block i.

The function $E()$ is computed by multiplying the percentage of changes, for each operation (Op_j) and block Bk_i, by the importance of operations $I(Op_j)$ and blocks $I(Bk_i)$. It returns a normalized value between 0 and 1.

Example 1.
Given the blocks importance as shown in Figure 7, the change operations detected in a delta are: (i) an update of a text in block B_1; (ii) an insertion of 4 links in block $B_{2.2}$; (iii) a deletion of 2 images in block B_3. In this paper, as mentioned in Section 5, we do not deal with changes in the block structure. Only the content of block is modified from one version to another. In this example, the operations insert and update are considered more important ($I(ins) = I(upd) = 1$) than a delete ($I(del) = 0.8$). In the old version of the page, the block B_1 has one text element, $B_{2.2}$ has 2 links and B_3 has 4 images. The importance of changes is computed as follows:
$E = I(bk_1) * I(upd) * [N(upd, text)/N(text, bk_1)] + I(bk_{2.2}) *$

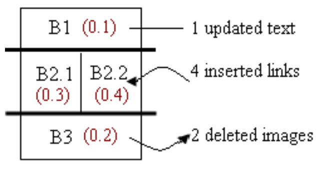


Figure 7: Block Importance Example.

$I(ins) * [N(ins, link)/N(link, bk_2)] + I(bk_3) * I(del) * [N(del, image)/N(image, bk_3)] = 0.1 * 1 * (1/1) + 0.4 * 1 * (4/(2+4)) + 0.2 * 0.8 * (2 /4) = 0.44$

This function $E()$ can be used by the scheduler to choose the most urgent document to be refreshed according to the history of changes. As mentioned in Section 3.1, the scheduler manages a list of documents ordered by a freshness urgency function. This function takes into account the importance of changes (estimated by function $E()$) that have occurred between the original version and the last one archived. We are currently working to define a best freshness urgency function for the scheduler that uses $E()$ as estimator of changes importance. We expect that this function improves the event-driven crawler by retrieving the most urgent document from the web. In further work, we hope also to use this function to compute the best frequency for the first type of crawler (the frequency crawler). Also, others parameters can be used to evaluate the importance of changes such as the page rank, the depth where a page is in a site (e.g., deeper a page is, the less important it is), etc.

7. IMPLEMENTATION AND VALIDATION

To evaluate the feasibility of our approach and to analyze its performance, experimental studies were performed. We first present the evaluation of our visual segmentation methods over HTML pages from the web. Then, results obtained by running our Vi-DIFF algorithm over various Vi-XML documents are analyzed.

7.1 Visual Segmentation

Visual segmentation experiments have been conducted over HTML web pages by using the extended VIPS method. We measured the time spent by the extended VIPS to segment the page and to generate the Vi-XML document. We present here results obtained over various HTML documents sized from about 20 KB up to 600 KB. These represent only sizes of container objects (CO) of web pages. A container object (CO) is usually a HTML file that references external objects (EO) like images, video, etc. According to Andrew King's research [20], the average size of web pages was about 312 KB (50 % of it is the size of CO) in 2008. Thus, the CO's size of the average web page is about 156 KB that we can use as reference to analyze our results.

We measured the performance of the visual segmentation in terms of execution time and output size. The implementation of the visual segmentation was written in the C++ programming language by extending the dynamically linked library VIPS.DLL [8]. Experiments were conducted on a PC running Microsoft Windows Server 2003 over a 3.19 GHz Intel Pentium 4 processor with 6.0 GB of RAM.

The execution time for the browser rendering and the visual segmentation is shown in Figure 8. The horizontal axis represents the size of HTML documents in KBytes (KB), and the vertical axis shows the execution time in seconds for each document. The time for rendering is almost constant, about 0.1 seconds. The execution time of the visual segmentation increases according to the size of HTML documents. The average time of the segmentation is about 0.2 seconds for documents sized about 150KB. This execution time seems to be a little bit costly but is counterbalanced by the richness of the Vi-XML file that really simulates the visual aspect of web pages. Nevertheless, this time cost must be optimized. The main idea for that purpose is to avoid rebuilding the blocks structure for a page version if no structural change has occurred since the former version. We are currently trying to find a method that detects directly changes inside blocks based on the visual structure of previous versions of the document.

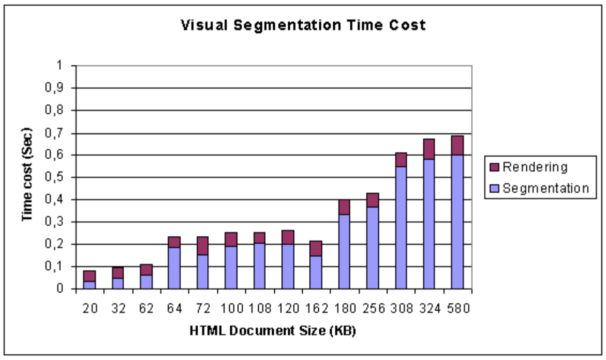


Figure 8: Segmentation Time

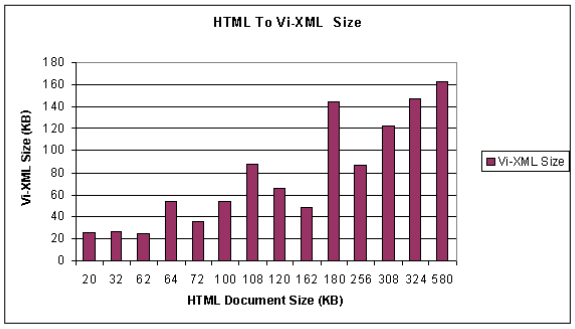


Figure 9: HTML TO Vi-XML

Figure 9 presents the size of the output Vi-XML file with respect to the size of the original HTML document. From this experiment, we can observe that the Vi-XML document size is usually about 30 to 50 percent less than the size of the original HTML document (for those sized more than 100 KB). This is interesting for the comparison of two Vi-XML documents since it can help to reduce the time cost of changes detection algorithm. In the next section, we use the generated Vi-XML documents corresponding to Figure 9 to experiment our Vi-DIFF algorithm.

7.2 Vi-DIFF

Experiments were conducted to analyze the performance of our proposed Vi-DIFF algorithm in terms of execution time and delta size. The implementation is written in Java programming language. Tests were conducted on PC running Linux over a 3.20 GHz Intel Pentium 4 processor with 1.0 GB of RAM. To better analyze the performance of our Vi-DIFF, we build a simulator that generates synthesized changes on given Vi-XML documents. The simulator takes a Vi-XML file and it generates a new version according to the parameters given as input (proportion of leaf blocks changed, for each operation type). It also generates the corresponding delta file that helps checking if the result of the Vi-DIFF is correct. We have tested our algorithm with the same Vi-XML documents as in Figure 9. 10 % of changes are fixed for each operation type (insert, delete, update, move) for links and images. For texts, only an update operation is considered. We have analyzed precisely each different time

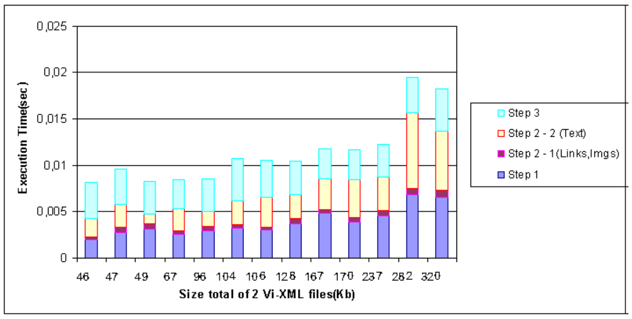


Figure 10: Vi-DIFF Execution Time

spent in main functions: Initialization (Step1), change detection for links and images (Step 2-1), change detection for texts (Step 2-2) and save of delta file (Step 3). As shown in Figure 10, change detection for links and images is the fastest part of the whole process. Initialization increases with the size of files because the more links (image, text) we have, the more objects we must create. The time of step 2-2 does not depend on files size but the length of texts in it. The total execution time is satisfying as it allows to process more than one hundred (currently sized) pages per seconds and per processor. Others tests have been realized to measure the size of output delta. Results show that the size of the delta is always less than the size of one version of Vi-XML document, which is reasonable. To check the correctness of the delta, we manually compared the delta generated by the simulator with the delta produced by our Vi-DIFF, with success.

8. CONCLUSION AND FUTURE WORKS

In this paper, we pointed out the issue of efficiently archiving web pages. Web archiving can waste time and space for indexing and storing unimportant changes on page versions. In our context (repository for the French INA), the visual aspect of pages is the most important to preserve. Thus, our approach is based on the visual representation of pages to better detect important changes between versions. Other approaches for Web archiving are based only on the frequency of changes.

The first step of our approach consists of constructing the whole visual layout structure of document based on semantic blocks. To this end, we extended VIPS [8], an existing algorithm for page segmentation, by adding features to the segmented document. Those new features are useful for the subsequent changes detection phase. This latter consists of comparing the last version archived of a page and the former one. The Vi-Diff algorithm we designed for this phase is more adequate to the visual layout structure of documents than existing generic methods. Then, issues related with evaluating the importance of changes were discussed. We expect this evaluation to be used, in a future step, to better predict the frequency of crawlers.

Preliminary tests on the segmentation and diff phases show that the execution time is promising. However, the time for segmentation is much higher than the time for comparing. In order to further optimize the system, we must focus on reducing the segmentation time. One idea is to assume that the block structure is evolving very rarely. Thus we can try to parse directly the HTML code, to check if the structure has changed or not. If not, then the new values for the blocks attributes can be extracted without segmentation. The idea is to process the segmentation only when a change in the structure is detected. We are currently investigating how to detect such a change in the structure by just parsing the HTML code. Another, yet complementary, idea is to rewrite the segmentation module from scratch. Indeed, the actual version is an extension of an existing code, which moreover uses a browser to get the rendered page. By using an ad'hoc rendering engine and recoding the whole module, we hope to reduce the segmentation time drastically.

Another on-going work is the handling of changes in the block structure. Not only we must detect those changes, but also extend the Vi-Delta XML structure to store them. Also, we intend to propose a visualization of the Vi-Delta in order to help user to better assess the importance of changes.

Future works are related to the urgency function and importance estimation. We are currently looking for the best machine learning technique to get automatically the relative importance of blocks and of change operations. The used technique should take into account the importance of contents in blocks especially when affected content can have a high significance according to the context of the page.

9. REFERENCES

[1] Internet Archive Wayback Machine, http://www.archive.org.

[2] A National Library of Australia Position Paper. National strategy for provision of access to australian electronic publications. www.nla.gov.au/policy/paep.html.

[3] The Web archive bibliography, http://www.ifs.tuwien.ac.at/ aola/links/webarchiving.html.

[4] S. Abiteboul, G. Cobena, J. Masanes, and G. Sedrati. A First Experience in Archiving the French Web. In *ECDL '02: Proceedings of the 6th European Conference on Research and Advanced Technology for Digital Libraries*, 2002.

[5] H. Artail and K. Fawaz. A fast HTML web page change detection approach based on hashing and reducing the number of similarity computations. *Data Knowl. Eng.*, 66(2):326–337, 2008.

[6] K. P. Arvidson and J. Mannerheim. The kulturarw3 project - the royal Swedish web archiw3e - an example

of 'complete' collection of web pages. In *66th IFLA Council and General Conference*, 2000. www.ifla.org/IV/ifla66/papers/154-157e.htm.

[7] D. J. C. Lampos, M. Eirinaki and M. Vazirgiannis. Archiving the greek web. In *4th International Web Archiving Workshop (IWAW04)*, Bath, UK, 2004.

[8] D. Cai, S. Yu, J.-R. Wen, and W.-Y. Ma. VIPS: a Vision-based Page Segmentation Algorithm. Technical report, Microsoft Research, 2003.

[9] W. Cathro. Development of a digital services architecture at the national library of Australia. EduCause, 2003.

[10] S. S. Chawathe and H. Garcia-Molina. Meaningful change detection in structured data. In *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, New York, NY, USA, 1997.

[11] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, New York, NY, USA, 1996.

[12] J. Cho and H. Garcia-Molina. The Evolution of the Web and Implications for an Incremental Crawler. In *VLDB '00: Proceedings of the 26th International Conference on Very Large Data Bases*, 2000.

[13] J. Cho and H. Garcia-Molina. Estimating frequency of change. *ACM Trans. Interet Technol.*, 3(3), 2003.

[14] G. Cobéna, T. Abdessalem, and Y. Hinnach. A comparative study for XML change detection. Technical report, 2002.

[15] G. Cobena, S. Abiteboul, and A. Marian. Detecting changes in XML documents. In *ICDE '02: Proceedings of 18th International Conference on Data Engineering*, 2002.

[16] C. N. Cosulschi M. and G. M. Classification and comparison of information structures from a web page. In *The Annals of the University of Craiova*, 2004.

[17] M. K. Evi, M. Diligenti, M. Gori, M. Maggini, and V. Milutinovi. Recognition of Common Areas in a Web Page Using Visual Information: a possible application in a page classification. In *the proceedings of 2002 IEEE International Conference on Data Mining ICDM'02*, 2002.

[18] D. Gomes, A. L. Santos, and M. J. Silva. Managing duplicates in a web archive. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, 2006.

[19] X.-D. Gu, J. Chen, W.-Y. Ma, and G.-L. Chen. Visual Based Content Understanding towards Web Adaptation. In *Second International Conference on Adaptive Hypermedia and Adaptive Web-based Systems (AH2002)*, 2002.

[20] A. B. King. *Website optimization*. O'Reilly, 2008.

[21] R. La-Fontaine. A Delta Format for XML: Identifying Changes in XML Files and Representing the Changes in XML. In *XML Europe*, 2001.

[22] E. Leonardi, T. T. Hoai, S. S. Bhowmick, and S. Madria. DTD-Diff: A change detection algorithm for DTDs. *Data Knowl. Eng.*, 61(2), 2007.

[23] T. Lindholm, J. Kangasharju, and S. Tarkoma. Fast and simple XML tree differencing by sequence alignment. In *DocEng '06: Proceedings of the 2006 ACM symposium on Document engineering*, 2006.

[24] L. Martin. Networked electronic publications policy, 1999 .www.nlc-bnc.ca/9/2/p2-9905-07-f.html.

[25] M. Naftalin and P. Wadler. *Generics and Collections in Java*. O'Reilly, 2005.

[26] L. Peters. Change detection in XML trees: a survey. In *3rd Twente Student Conference on IT*, 2005.

[27] S. R. Singh. Estimating the rate of web page updates. In *IJCAI*, 2007.

[28] R. Song, H. Liu, J.-R. Wen, and W.-Y. Ma. Learning block importance models for web pages. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, 2004.

[29] Y. Wang, D. DeWitt, and J.-Y. Cai. X-Diff: an effective change detection algorithm for XML documents. In *ICDE '03: Proceedings of 19th International Conference on Data Engineering*, March 2003.

[30] Y. Yang and H. Zhang. HTML Page Analysis Based on Visual Cues. In *ICDAR '01: Proceedings of the Sixth International Conference on Document Analysis and Recognition*, 2001.