Tracking Changes in Healthcare Documents*

Sudarshan S. Chawathe
Department of Computer Science, University of Maine, Orono, Maine 04469, USA
chaw@cs.umaine.edu

Abstract

We present methods for monitoring a large, diverse, and autonomously modified collection of healthcare documents on the Web. Our methods do not require document-providers to offer any special services. They are based on explicating changes between document versions on a per-user basis by using differencing algorithms. These changes are presented to users in the context of the documents using special XML elements. In order to effectively browse changes in large document collections, we use a variable-resolution XML browser. A noteworthy feature of this browser is that it produces usable displays at any level of detail specified by a user.

1 Introduction

Healthcare information on the Web is growing in both size and scope. The available information covers diverse aspects of the field, such as research results, governing regulations from several bodies, and resources for patients and healthcare providers. This information is distributed not only physically, residing on servers worldwide, but also logically, being controlled by diverse, autonomous organizations, such as government agencies at federal, state, and local levels, professional organizations, and private groups. This decentralization of control over the information is responsible, in large part, for its growth, and for the success of the Web in general.

However, this decentralization also creates several data-management difficulties. The decentralization results in heterogeneity at levels ranging from data representation (e.g., character sets and notation) to semantic (ontologies), making it difficult to extract consistent information from the distributed resources. Further, the information evolves in a rapid and unpredictable manner. Thus, even if we overcome the problems of heterogeneity and arrive at a consistent view of the information we seek, it is difficult to stay current and to separate the important and urgent changes from those that can be ignored or processed later. In this paper, we focus on the latter problem: How can we assist a healthcare professional with the task of effectively keeping track of the relevant body of Web documents when the documents are being continually updated in unpredictable ways by their autonomous owners?

The need for allowing readers to track changes in documents they have read, as well as to new documents that may be of interest has been recognized by many information providers, as evidenced by "what's new" sections often found on Web sites. For instance, the site from which we draw our running example (described in Section 2) includes such a section [4]. There are, however, two problems with these efforts: First, not all providers offer such a service. There is also a great variety in the manner and frequency of such information. Second, and more important, the interpretation of "new" is determined by the information provider, and not the reader. This interpretation is presumably based on some notion of a typical reader or an expected frequency of visits to a Web site. A reader who visits a Web site frequently is likely to find old information still marked as new, whereas a reader who visits infrequently is likely to miss changes that are removed from the "what's new" section between visits. Our approach in this paper avoids such problems by providing a per-user description of changes. The resources required for this task, such as storage space and computational power, are based at the user's site (e.g., personal computer) instead of at the server sites. Therefore, this approach scales well as the number of users grows.

In Section 2 we introduce our running example and describe the first stage of our method: explicating changes between versions of a document, such as the versions encountered on two visits to a Web site. Our solution is based on formalizing the problem as one of finding a minimum-cost edit script between two trees representing versions of an XML document. The output of this stage is an edit script that encodes the precise changes between versions. This information is derived without any additional features from the document providers.

Although an edit script is a precise representation of changes between versions of a document, it is better suited for interpretation by machines than by humans. In Section 3, we describe the second stage of our method: presenting the changes found by the first stage. Our solution is based on embedding an edit script in a recent version of the document being monitored, allowing changes to be viewed

*This work was supported in part by the U.S. National Science Foundation with grants IIS-9984296, IIS-0081860, and CNS-0426683.

Questions	Assessment Are an of the following present?
Age of the patient	• >65 • <65 with co-morbidity • child <12 (Note —zanamivir not recommended in this age group). May or may not require referral to a doctor depending on overall condition
Pregnancy • Are you pregnant or breast feeding?	• Pregnant or unsure • Breast feeding
Duration • When did symptoms start?	• <36 Hours • >36 hours but <1 week • > 1 week
Previous history / co-morbid conditions • Are you normally fit and well?	Any of the following conditions? • Asthma requiring regular medication

Figure 1. *An excerpt of a document outlining a triage protocol for flu-like illnesses [12]. An XML rendering appears in Figure 2.*

```
<site> <fordoctors>
   <protocols> <clinical> <protocol>
      <title>Telephone triage for flu-like
      illness</title>
      <form-item>Name of caller</form-item>
      <form-item>Name of patient, if different
      </form-item>
      <note>Look at patient medical notes to
      supplement the information provided by
      patient.</note>
      <qblock category="Age of the patient">
         <assessment id="00201">
            <answer code="blue" id="00231">
            Greater than 65.
            </answer>
            <answer code="blue">
            Less than 65 with co-morbidity.
            </answer>
            <answer code="blue">
            Child less than 12.
               <note>Zanamivir not recommended
               in this age group.</note>
            </answer>
       <note>May or may not require
               referral to a doctor depending
       on overall condition.</note>
         </assessment>
      </qblock> ...
   </protocol> ... </clinical> ... </protocols>
... </fordoctors> ... </site>
```

Figure 2. *XML representation of a medical information site [4], with details corresponding to the document of Figure 1.*

and interpreted in the appropriate context.

However, as the number of changes increases, or as the documents being monitored get larger, this method for monitoring changes may present more information than is easily processed. In Section 4, we describe the third stage of our method: providing a variable-resolution interface to documents and their changes. Our solution is based on an XML browser, called *FuzzyTree*, that summarizes data at varying levels of detail. By preprocessing the output of the previous stage, *FuzzyTree* permits multi-resolution browsing of documents and changes with interactive response times. We discuss related work in Section 5 and conclude in Section 6.

2 Changes in Documents

Figure 1 depicts a small excerpt from a document describing the protocol for triage of patients with flu-like symptoms [12]. An XML rendering of this document is suggested by Figure 2, in which we use ellipses (`...`) to denote additional data that has been omitted for brevity.

Suppose the document of Figure 2 is modified by its issuing authority, perhaps to reflect changes in the protocol prompted by recent research. Figure 3 suggests such a modified version (hypothetical) of the document. As indicated by the line marked with a `!`, the code for the answer "greater than 65" has been changed from blue to red. Further, the lines marked `+` indicate newly inserted data: a procedure for ages 16–22, along with a qualifying note. We note that the `!` and `+` marks are for ease of presentation only and are not part of the document.

Thus, we have the following problem: Given two document versions, such as those suggested by Figures 2 and 3, how do we determine precisely what has changed between versions? In order to formalize this problem, we model the XML documents as rooted, ordered, labeled trees. This model is widely used, with the trees representing DOM interpretations of the XML documents [8]. Figure 4 suggests a few such trees. Tree nodes are represented as circles that contain text representing the node's value (XML content). To simplify presentation, we do not distinguish between XML attributes and XML content of various types; such distinctions, along with others, such as typing, are easily added.

Changes to XML documents are modeled as *edit operations* on the corresponding trees. We use a simple edit model consisting of three operations: (1) insertion of a tree node, (2) deletion of a node, and (3) update of a node's label. The effect of these operations is suggested by the arrows in Figure 4. The notation and semantics of the edit operations are summarized below:

upd(n, v)	Update the label of node n to v.
del(n)	Delete node n, which must not have any descendants.
ins(n, p, c, v)	Insert a node n labeled v as the c'th child of node p, which must exist and have at least $c - 1$ children.

A sequence of edit operations is called an edit script; its action on a tree is the result of applying its constituent operations, in sequence. In Figure 4, proceeding from the top-left tree T_1 to the bottom-left tree T_2 along the arrows depicts the result of applying the edit script *upd*$(6, B)$, *upd*$(8, F)$, *del*(8), *upd*$(6, C)$.

```
 ... ... ...
       <qblock category="Age of the patient">
          <assessment>
!            <answer code="red">
             Greater than 65.
             </answer>
             <answer code="blue">
             Less than 65 with co-morbidity.
             </answer>
+            <answer code="blue">
+            Between 16 and 22.
+            <note>Especially college students.</note>
+            </answer>
 ... ... ...
```

Figure 3. *A modified version (hypothetical) of the document of Figure 2. The ellipses denote several unchanged lines at the beginning and end of the document that have been omitted for brevity. The "!" and "+" indicators that highlight the differences are for our presentation here only, and are not part of the document.*

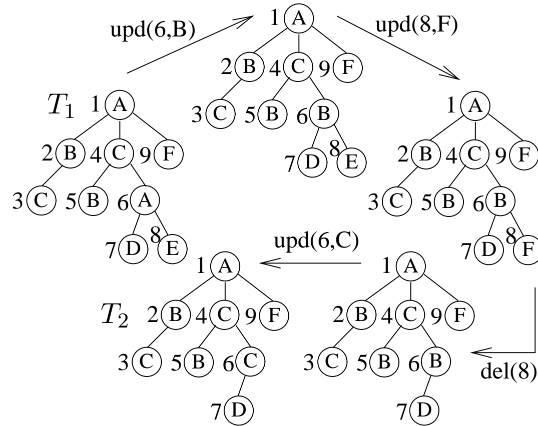

Figure 4. *Edit operations for describing changes in hierarchical documents, such as the one in Figure 3.*

The problem of explicating changes in XML is then the problem of discovering such an edit script that transforms the tree representing the old version of an XML document into the tree representing the new version. For example, we may be given the top-left and bottom-left trees, T_1 and T_2, of Figure 4 as input and be asked to determine the edit script. We note here that, given any two trees, there are many edit scripts that transform one to the other, but some are intuitively more desirable than others. For instance, we may always use an edit script consisting of operations to delete all nodes in the first tree followed by operations to insert the nodes in the second tree. Such an edit script is intuitively undesirable because it seems to include unnecessary operations. We formalize this notion by defining the cost of an edit script to be the number of operations it contains and seeking a minimum-cost edit script.

In our example of Figure 4 it is easy to verify that the edit script $upd(6, C), del(8)$ is a minimum-cost edit script that transforms the top-left tree to the bottom-left one. The $upd(6, B)$ and $upd(8, F)$ operations in the edit script discussed earlier are unnecessary because node 6 is updated again later, while node 8 is deleted later. In larger examples, such as those arising in the scenario suggested by Figures 2 and 3, determining a min-cost edit script is not this easy, but has been the subject of prior work. For example, we may use Selkow's method [13], or a modification for large datasets [1] based on the Myers's technique of computing along diagonals [11]. For our running example, such methods yield the following as an edit script that transforms the tree of Figure 2 into that of Figure 3:

```
upd(00231,code,"red")
ins(01001,00201,1,<answer code="red">)
ins(01002,01001,1,"Between 16 and 22.")
ins(01003,01002,1,<note>)
ins(01004,01003,1,"Especially college students.")
```

In the above edit script, the node identifiers refer to `id` attributes in our XML data. For example, the first insert operation indicates that a new node with `id` 01001 (the first line marked + in Figure 3) is inserted as the first child of a preexisting node with `id` 00201 (the first `assessment` node in Figure 2) . These identifiers are analogous to the numbers next to the nodes in Figure 4. They are, in general, implementation-specific and their values are not predictable. In particular, we cannot use such identifiers to reliably find matching nodes in old and new versions of a document.

3 Presentation of Differences

Although our method of Section 2 allows us to determine the precise changes made to a document since the last time we viewed it, the form in which these changes are presented is not convenient. For our running example, the output of this method is the edit script outlined at the end of the previous section. Although the changes in our running example are few and intuitively easy to describe, the changes encoded by the edit script are difficult to decipher, even in conjunction with Figures 2 and 3 (for determining the meaning of the node identifiers). When we consider the larger documents and more numerous changes expected in our motivating scenarios, it is unreasonable to expect users to decipher such edit scripts.

An appealing solution to the above problem consists of, intuitively, embedding the edit script in the new version of the document itself, as suggested by Figure 5 for our running example. The edit script of the previous section is embedded using new XML attributes and elements that use a namespace denoted `d` in the figure. Following usual conventions, such a namespace identifier is bound to a longer, glob-

```
 ... ... ...
       <qblock category="Age of the patient">
          <assessment>
*            <answer code="red"
*             d:aup="code" d:aov="blue">
             Greater than 65.
             </answer>
             <answer code="blue">
             Less than 65 with co-morbidity.
             </answer>
*            <answer code="blue" d:ins>
*            <d:itx/> Between 16 and 22.
*            <note d:ins><d:itx/>Especially
*               college students.</note>
             </answer>
 ... ... ...
```

Figure 5. *The document of Figure 3 with annotations, in the XML namespace* d*, indicating differences from the version of Figure 2. The ellipses mark omitted lines, as in Figure 3. The "*" indicators are for our presentation here only, and are not part of the document.*

ally unique string in the XML preamble, which we omit. The affected lines are marked with * for ease of presentation. The `d:aup="code"` annotation on the first answer element indicates that the code attribute of that element was updated. The old value is preserved as the value of the `d:aov` attribute that follows. Inserted elements are annotated with a `d:ins` attribute, as is the third answer element, and the note element it contains, in Figure 5. Finally, inserted text is marked using a `d:itx` element.

The XML embedding of edit scripts suggested above is suitable for displaying changes to documents in context and frees users from having to decode edit scripts using node identifiers. However, it does not scale very well as the number of changes, and size and number of documents, increases. For example, consider a document that is several hundred pages in length and that has been modified in dozens of places all over the document. Using the method described so far, we can detect the precise changes and present the user with a marked-up version of the document, perhaps using different colors and fonts to mark changes and improve readability. However, the user is still forced to browse the entire document and visually scan for changes. The situation is worse if there are several such documents. In addition to the increased demands on the user's attention and time, very large documents and large numbers of changes also pose implementation problems for browsing tools. In the next section, we present a method for coping with large document collections and large numbers of changes.

4 Variable-Resolution Interfaces

We may state the problem outlined at the end of the previous section as follows, continuing with our model, from Section 2, of XML documents as rooted, ordered, labeled trees: We are given as input a typically large tree and must produce as output a *representative substructure* that *best represents* the input tree at a specified *level of detail*. In order to complete the problem definition, we now define each of the italicized phrases. We define a representative substructure to be a tree that is a connected subgraph of the input tree, with the same root. (It is possible to conceive of other options, such as permitting disconnected subgraphs; however, our preliminary experiments reveal that this definition results in structures that are intuitively easier to map to the original data than are others.) We define the level of detail to be the number of nodes in the representative substructure. The definition of the phrase *best represents* is the one most open to alternatives. Our operational definition, presented below, is one we have found to yield usable results on a variety of datasets from diverse domains.

Figure 6 is a screen-shot of *FuzzyTree*, our implementation of a variable-resolution XML browser, displaying the document suggested by Figure 5. For this test, we used the complete document, not the small excerpt suggested by earlier figures. By design, the *FuzzyTree* browser requires the user to manipulate only one parameter: the level of detail, specifically, the number of nodes (XML elements) in the display. This parameter is set using the slider that appears near the top of the browser window. The number of elements displayed, as well as the total number of elements in the file, are reported by *FuzzyTree* below the slider. The level of detail may also be entered directly using a text box that appears below the slider. This feature is useful when the range of the slider is very large and large moves are required.

The main part of the interface is an extension of Java's standard *JTree* interface [9], which in turn is similar to classic Macintosh *Finder* interface in details mode. In a JTree interface, a tree structure, such as an XML DOM tree or a filesystem hierarchy, is depicted using a horizontal layout of the tree. An interior node of this tree may be in either an open state, in which case its children are displayed, or a closed state, in which case its children, and all descendants, are hidden. A user may toggle between these states for a node by clicking on its icon. *FuzzyTree* adds a third state, which we call half-open, in which some, but not all, of a node's children are displayed. In Figure 6, the site, protocol, qblock, and assessment nodes are in this state, as indicated by the node icons with indicator lines at 45-degree angles. The assessment node, for example, has only four of its five children displayed. Clicking on a node in the *FuzzyTree* interface has semantics different from those

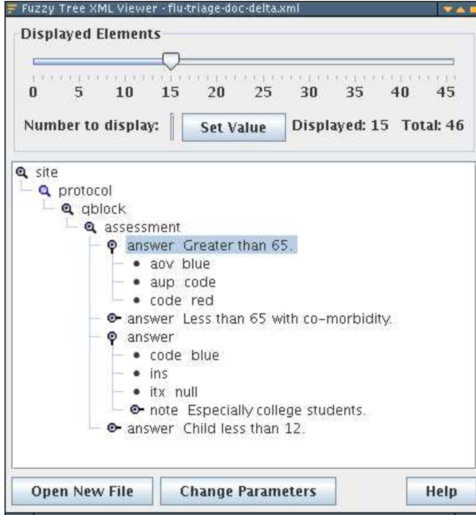

Figure 6. *A screen-shot of the* FuzzyTree *variable-resolution browser operating on the document of Figure 5. The selective display algorithm exposes the modifi ed data (aov, aup, ins, etc.) while hiding other, less important, data.*

in JTree. Instead of instantly opening or closing the corresponding node, such an action may be thought of as gently suggesting greater or lesser interest in that node, and thus affects the display indirectly. More precisely, clicks on node icons control the third term in a node's display score, which is described later in this section. The only remaining features of the interface are the buttons near the bottom for loading a new file and viewing online help. (The *Change Parameters* button at the bottom of the window sets parameters meant mainly for debugging.)

We note that the *FuzzyTree* display of Figure 6, although limited to 15 nodes (an artificially low limit for purposes of illustration), conveys nicely the overall structure and the most important parts of the document of Figure 5. We now describe the method used by *FuzzyTree* to select nodes that achieve such a result in general. Each node is assigned a score that is the sum of three terms, which are described below. The D highest-scoring nodes are displayed by *FuzzyTree*, where D is the level-of-detail parameter set using the slider.

The first term in the score is based on structural isomorphism and is a measure of how many other nodes in the tree are structurally similar to it. In more detail, let $t(n)$ denote the subtree rooted at a node n and let $|t(n)|$ denote the number of nodes in $t(n)$. Let $I(t)$ denote the set of subtrees of the input tree that are isomorphic to a tree t. Then the first term of n's score is $|t(n)| \cdot |I(t(n))|$. Isomorphism is defined recursively as follows: (1) Two leaf nodes are isomorphic if they have identical content. (2) Two interior nodes are isomorphic if they have the same number of children and, further, if their corresponding children are isomorphic, recursively.

The second term in a node's score is based on the results of Section 2. Nodes that are annotated in order to mark changes, as well as nodes representing the annotations themselves, such as `<d:itx/>`, have a value K for this term, while others have value 0. The parameter K determines the significance of this term in relation to the first.

The third term in a node's score, unlike the other two, is not fixed based on the input; rather, it's value depends on user actions during browsing. (Recall our description of the *FuzzyTree* interface above.) Initially, this term is zero for all nodes. When the user clicks on a node that is closed or half-open, this term for that node is incremented. When a user clicks on a fully open node, this term is decremented. This interpretation of user clicks provides a gently changing display that we have found to be very useful in browsing large documents. Nevertheless, a large increment or decrement may be achieved in a single click using a shift-click combination, with the effect of immediately fully opening or fully closing a node.

5 Related Work

Vasilyeva et al. emphasize the need for adaptive user interfaces for healthcare information systems and present a framework for such systems [14]. Curé describes the XIMSA system for providing information to patients to aid self-medication [6]. XIMSA uses information from a simplified electronic health record and a knowledge base with a suitable ontology. The above methods share many goals with our work in this paper and it would be interesting to explore how they may be combined. Ciampolini et al. address the problem of verifying compliance of medical systems with medical protocols on-the-fly, as the system operates, by analyzing the stream of events [5]. Their method is based on social integrity constraints expressed using a logic-based formalism. Our *FuzzyTree* system may be used to present the results of such compliance verification to an administrator, with node scores representing, in part, the seriousness of violations. Lau et al. use structural features of documents to analyze relationships among regulations from diverse sources, such as regulations for disabled persons' access from federal, state, and local governments [10]. Their study underscores the complexity of such documents and the need for tools that help users discover relevant information and monitor changes. Combining some of their techniques, such as the structure-based similarity detection, with our methods for differencing and browsing should be a promising direction for further work.

Several forms of the general problem of differencing data

have been studied. GNU diff [7] is a popular differencing program that is used in many applications, such as version control and file merging. This program treats the input files as sequences of lines (by default) and uses Myers's algorithm [11]. Although such sequence-comparison methods may be applied to hierarchical XML data as well, the use of tree-differencing methods yields results that are more meaningful in the context of hierarchical data. An early method is Selkow's dynamic programming solution [13] using edit operations very similar to those used in this paper. Recent work has focused on improving efficiency on large datasets and streams [1, 2]. Further, enhanced edit models, such as those including operations that move and copy subtrees [3] have also been studied. In general, such work complements the general method presented in this paper by providing more nuanced descriptions of changes in documents. However, such improvements often come with substantial increases in running time.

6 Conclusion

We motivated the need for tools that enable healthcare professionals, as well as patients, to monitor a large, diverse, and evolving collection of documents on the Web. Of the many challenges presented by the Web environment in this context, we focused on two in this paper: How to explicate changes in Web documents and how best to present these changes to users.

Our solution to the first problem is based on comparing each document viewed by a user with the version of that document when the user last viewed it. These documents are stored and managed on the user's computer, providing each user with a customized "what's new" feature that does not depend on the document-provider. Our document comparison method is based on formalizing the problem as that of computing a minimum-cost edit script between two rooted, ordered, labeled trees representing the documents.

Our solution to the second problem is based on (1) embedding changes (edit scripts) in documents using special XML markup and (2) an adaptive method for selective display of nodes in tree-structured data, as implemented in *FuzzyTree*. A notable feature of *FuzzyTree* is that it provides a meaningful summary of its input at a user-specified level of detail. In order to determine the nodes best suited for this purpose, it uses a combination of subtree isomorphism, changes, and runtime history of browsing. The level of detail can be smoothly modified using a slider, with interactive response times.

In continuing work, we are extending *FuzzyTree* to improve its efficiency on extremely large disk-resident datasets, such as outputs of simulation experiments. We are also extending it to streaming data, in which case the display represents highlights of data seen so far. In addition, we are investigating alternate, domain-specific methods of explicating differences between document versions in order to produce better descriptions of changes.

Acknowledgment Major parts of the *FuzzyTree* system were implemented by Donna Malayeri, with subsequent modifications by Nicholas Kleinschmidt.

References

[1] S. S. Chawathe. Comparing hierarchical data in external memory. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 90–101, Edinburgh, Scotland, Sept. 1999.

[2] S. S. Chawathe. Differencing data streams. In *Proceedings of the 9th International Database Engineering and Applications Symposium (IDEAS)*, pages 273–284, Montreal, Canada, July 2005.

[3] S. S. Chawathe and H. Garcia-Molina. Meaningful change detection in structured data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 26–37, Tuscon, Arizona, May 1997.

[4] B. Cheek. The gp-training.net project. `http://gp-training.net/`, Mar. 2005.

[5] A. Ciampolini, P. Mello, M. Montali, and S. Storari. Using social integrity constraints for on-the-fly compliance verification of medical protocols. In *Proceedings of the IEEE Symposium on Computer-Based Medical Systems (CBMS)*, pages 503–505, Dublin, Ireland, June 2005.

[6] O. Curé. Ontology interaction with a patient electronic health record. In *Proceedings of the IEEE Symposium on Computer-Based Medical Systems (CBMS)*, Dublin, Ireland, June 2005. 185–190.

[7] M. Haertel, D. Hayes, R. Stallman, L. Tower, P. Eggert., and W. Davison. The GNU diff program. Texinfo system documentation, 1998. Available through anonymous FTP at `prep.ai.mit.edu`.

[8] A. L. Hors, P. L. Hgaret, L. Wood, G. Nicol, J. Robie, M. Champion, and S. Byrne. Document Object Model Level 2 Core Specification. W3C Recommendation, W3C, `http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113`, Nov. 2000.

[9] Java 2 platform, standard edition, v 1.4.2 API specification. Sun Microsystems. `http://java.sun.com/`, 2003.

[10] G. T. Lau, K. H. Law, and G. Wiederhold. Analyzing government regulations using structural and domain information. *IEEE Computer*, 38(12):70–76, Dec. 2005.

[11] E. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986.

[12] Northumberland Health Authority. Telephone triage for flu-like illness. `http://www.gp-training.net/protocol/infections/flutriage.htm`, Feb. 2006.

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

[14] E. Vasilyeva, M. Pechenizkiy, and S. Puuronen. Towards the frameworkof adaptive user interfaces for eHealth. In *Proceedings of the IEEE Symposium on Computer-Based Medical Systems (CBMS)*, pages 139–144, Dublin, Ireland, June 2005.