Dynamic Generation and Browsing of Virtual WWW Space Based on User Profiles

Norihide Shinagawa[1] and Hiroyuki Kitagawa[2]

[1] Doctoral Program in Engineering, University of Tsukuba,
Tennohdai 1-1-1, Tsukuba, Ibaraki 305-8573, Japan
siena@dblab.is.tsukuba.ac.jp

[2] Institute of Information Sciences and Electronics, University of Tsukuba,
Tennohdai 1-1-1, Tsukuba, Ibaraki 305-8573, Japan
kitagawa@is.tsukuba.ac.jp

Abstract. Due to the recent and rapid advance of the WWW technology, a huge amount of information is available via the Internet. However, it is difficult for users to find and utilize relevant information. Generally, querying and browsing are both indispensable to access to the required information. In this paper, we propose a scheme to support the browsing phase of the information search based on specification of the user's interest. In our scheme, each user is given virtual WWW pages reflecting his/her interest. Each virtual WWW page is dynamically generated based on the logical document structure extracted from the original page. Our approach makes it possible to browse the WWW space from his/her viewpoint and to identify required information more easily. We also show how such a scheme can be realized in a non-intrusive way in the current WWW browsing environment.

1 Introduction

Due to the recent and rapid advance of the WWW technology, a huge amount of information is available via the Internet. The number of available WWW pages is increasing incessantly, and it is difficult to find and utilize relevant information. A variety of schemes and tools have been investigated up to the present to address this issue [1] [2] [3].

Generally, both querying and browsing are required to access to and identify relevant WWW pages. In a typical scenario, first, the user gives queries to search engines and gets the set of candidate WWW pages. Then, he/she selects some from the candidate set and browses them. If necessary, he/she browses other linked pages and goes back to previous pages. In this manner, querying and browsing are used by turns to reach WWW pages which are relevant to the user's interest.

In the browsing process, the user's interest is usually implicit, while it is explicitly expressed as queries in the querying process. Therefore, it is usually the case that existing WWW pages are presented exactly in the same format and layout to all users, even if they have different interests. Thus, it is the user's responsibility to understand the structure and semantics of WWW pages and to identify relevant substructures and regions. However, it is not an easy task. In particular, in the WWW environment, the user has to manipulate a large amount of documents, most of which are unfamiliar to him/her. In addition, sometimes each WWW page contains a large amount of information. Of course, most of the WWW browsers provide the string search facility to help the user. However, simple string match is not enough to help him/her understand the page contents. Moreover, the string search facility does not change the situation that every user is given the same page presentations.

In this paper, we propose a new scheme to support the browsing process based on specification of the user's interest. In our scheme, the user's interest is explicitly expressed as a user profile. Then, virtual WWW pages named *view-pages* tailored to the user's interest are dynamically generated and presented when browsing the WWW. View-pages essentially show summaries of existing HTML pages complying with the user's interest and level of details specified in the user profile. Therefore, users of different interests are given different views of the WWW. In the view-page generation, a document model named a *logical tree* is used to model the logical structure of an HTML page and plays an important role. In this paper, we also show how view-pages can be realized in a non-intrusive way in the current WWW browsing environment. Therefore, the user can enjoy the proposed feature on the ordinary WWW browsers rather than being compelled to use special browsers tailored to this purpose.

The remaining part of this paper is organized as follows. In Section 2, we give an overview of our approach. In Section 3, we introduce a logical tree to cast logical hierarchies on HTML pages. In Section 4, we present a scheme to generate view-pages based on a user profile. The logical tree in Section 3 is used in the view-page generation. In Section 5, we show the system architecture to implement the proposed scheme non-intrusively in the ordinary WWW browsing environment. In Section 6, we mention related works. Finally, we summarize and conclude this paper in Section 7.

2 Overview

Here, we present an overview of the scheme proposed in the paper. Figure 1 illustrates our approach.

In our environment, the user's viewpoint is explicitly specified as a user profile. A user profile consists of keywords to represent the user's interest and a threshold value which controls the detail level of the presentation. Whenever the user accesses to WWW pages, view-pages are generated dynamically based on the user profile and presented to him/her. They may contain some hyperlinks referring to other WWW pages. In this sense, the user can navigate through the virtual WWW space. Each view-page is essentially a summary of the original WWW page from the viewpoint of the user's interest, and is generated as follows.

(1) We assume that WWW pages are written in HTML. In the first phase, a logical document structure embedded in the target HTML page is extracted, and

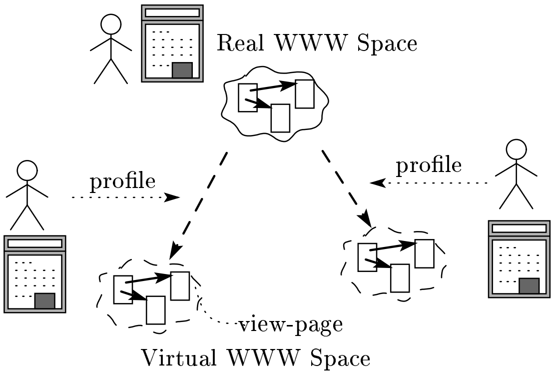

Fig. 1. Overview of the Proposed Scheme

is modeled in a document model named a logical tree. HTML pages contain tags, and some of them are used to construct the logical trees. However, as pointed in [2], the logical document hierarchy usually does not always coincide with the tag hierarchy.

(2) Similarity between each node in the logical tree and the user profile is calculated.

(3) We mark all nodes whose similarities are less than the threshold value in the user profile. A view-page is constructed by pruning away subtrees whose nodes and ancestors are all marked. This phase has to ensure that view-pages are also valid HTML documents.

The phase (1) is further explained in Section 3, and the phases (2) and (3) are elaborated in Section 4.

3 Logical Tree

In this section, we explain derivation of logical trees from HTML pages. For this purpose, first, we classify HTML tags in Subsection 3.1. Then, we show how tags in some group are used to derive logical trees in Subsection 3.2.

3.1 Classification of HTML Tags

We classify tags into the following four groups.

a) Structure-oriented tags: Tags in this group are primarily used to indicate document structures such as headings, paragraphs and lists of items. This group includes the following tags: `H1`, `H2`, `H3`, `H4`, `H5`, `H6`, `P`, `BLOCKQUOTE`, `DIV`, `UL`, `OL`, `DL`, `TABLE`. Generally, it is not easy to extract logical document structures from HTML documents, but tags in this group give clues to extract them. As a matter of fact, they are used to construct the logical tree as explained in Subsection 3.2.

b) Style-oriented tags: Tags in this group are mainly used to give special presentation effects. The tags `STRONG`, `EM`, `TT`, `I`, `U`, `B`, `BIG`, `SMALL`, `STRIKE`, `S`, `FONT`, `DL` are included in this group. Note that some of them are deprecated in HTML 4.0 [15], which recommends us to use style sheets instead. Elements enclosed by tags in this group are given predetermined weights when deriving feature vectors of document substructures as explained in Subsection 4.1.

c) Media-oriented tags: Tags in this group are used to embed media objects such as images and applets. They include the following tags: `IMAGE`, `FORM`, `SCRIPT`, `APPLET`, `OBJECT`, `EMBED`, `MAP`. They are ignored in the construction of logical trees. The contents of embedded media objects are also ignored in the derivation of feature vectors.

d) Miscellaneous tags: The other tags are ignored in the construction of logical trees. Existence of tags in this group does not have an effect on the derivation of feature vectors, and elements enclosed in those tags are treated as if they are ordinary character strings.

3.2 Derivation of Logical Trees

In this subsection, we describe derivation of a logical tree from an HTML document. Structure-oriented tags are used as important clues in the derivation. In some context, similar schemes to derive logical structures inherent in HTML documents are suggested (e.g. [2]).

A logical tree consists of the eight types of nodes: $\ll doc \gg$, $\ll desc \gg$, $\ll leading \gg$, $\ll trailing \gg$, $\ll packed \gg$, $\ll block \gg$, $\ll heading \gg$, $\ll paragraph \gg$. Given an HTML document, it is parsed according to rules in Figure 2, and the logical tree is obtained.

Each node of a logical tree has an attribute L, which indicates the hierarchy level of the node. The L-value of heading nodes associated with `H1`, `H2`, ..., `H6` tags are 1, 2, ..., 6, respectvely. Each rule in Figure 2 has an annotation "*attr*", which specifies the L-value of the node on the left-hand side. Some rules also have an annotation "*cond*" and the derived logical tree has to satisfy these conditions. Therefore, nodes closer to the root have smaller L-values in the logical tree, and, in this sense, L indicates the hierarchy level. **

In the case of $\ll desc \gg$, the "*cond*" says that the L-value of each non-$\ll leading \gg$ child node must be equal to each other, and the "*attr*" says that the L-value gives the attribute L of the $\ll desc \gg$ node.

Given the sample HTML document in Figure 3, the derived logical tree shown in Figure 4 is derived by the rules. The dotted lines indicate node groups having the same L-value.

The $\ll leading \gg$ nodes, which appear as the left-most child of $\ll desc \gg$ nodes, tend to give introduction or summary of the following $\ll trailing \gg$ nodes. This observation is used in the next section.

** Precisely speaking, exceptions may occur at nodes associated with `BLOCKQUOTE` tags.

$$
\begin{array}{rcl}
\ll doc \gg & \rightarrow & \texttt{<BODY>} \ll desc \gg \texttt{</BODY>} \\
attr & : & L := 0 \\
\ll desc \gg & \rightarrow & \ll leading \gg \ll trailing \gg + | \ll trailing \gg + | \ll paragraph \gg + \\
cond & : & \forall c_1, c_2 \; which \; are \; not \ll leading \gg \{ c_1.L = c_2.L \} \\
attr & : & L := c.L \; where \; c \; is \; not \ll leading \gg \\
\ll leading \gg & \rightarrow & \ll block \gg \\
cond & : & L < child.L \\
attr & : & L := parent.L \\
\ll trailing \gg & \rightarrow & \texttt{<DIV>} \ll trailing \gg \texttt{</DIV>} | \ll packed \gg \\
attr & : & L := child.L \\
\ll packed \gg & \rightarrow & \ll heading \gg \ll block \gg \\
cond & : & L < \ll block \gg .L \\
attr & : & L := \ll heading \gg .L \\
\ll block \gg & \rightarrow & \texttt{<DIV>} \ll block \gg \texttt{</DIV>} \\
 & | & \ll desc \gg \\
attr & : & L := child.L \\
\ll heading \gg & \rightarrow & \texttt{<H1>} text \texttt{</H1>} | \texttt{<H2>} text \texttt{</H2>} | \texttt{<H3>} text \texttt{</H3>} \\
 & | & \texttt{<H4>} text \texttt{</H4>} | \texttt{<H5>} text \texttt{</H5>} | \texttt{<H6>} text \texttt{</H6>} \\
attr & : & L := the \; level \; of \; the \; tag \\
\ll paragraph \gg & \rightarrow & \texttt{<BLOCKQUOTE>} \ll block \gg \texttt{</BLOCKQUOTE>} | \texttt{<DIV>} \ll block \gg \texttt{</DIV>} \\
 & | & \texttt{<P>} text \texttt{</P>} | \texttt{<UL>} text \texttt{</UL>} | \texttt{<OL>} text \texttt{</OL>} | \texttt{<DL>} text \texttt{</DL>} \\
 & | & \texttt{<TABLE>} text \texttt{</TABLE>} | text{-}without{-}structure \\
attr & : & L := \text{if tag is } \texttt{DIV} \text{ then } \ll block \gg .L \text{ else } 7
\end{array}
$$

Note 1: "*text*" is any character string, while "*text−without−structure*" accepts one that is not decomposable by any rule. Thus, "*text−without−structure*" accepts a string which appears immediately inside a `BODY`, `DIV` or `BLOCKQUOTE` element and does not contain any structure-oriented tags.
Note 2: "*node.L*" means the L-value of the *node*.

Fig. 2. Derivation Rules

4 Generation of View-pages

In this section, we explain how to construct view-pages based on the logical tree. We use the vector space model [4] [5] [6], but structure-oriented and style-oriented tags introduced in Subsection 3.1 are taken into consideration.

First, we derive a feature vector for each node in the logical tree. Second, we calculate the similarity between each node and the user profile using the feature vector. Next, we mark all nodes whose similarities are less than the threshold

```
<BODY>
   <H1> 1. heading of chapter </H1>
   <P> introduction of chapter </P>
   <H2> 1.1. heading of section </H2>
   <P> body 1.1.1 </P>
   <P> body 1.1.2 <B> IMPORTANT </B> </P>
   <H2> 1.2. heading of section </H2>
   <P> body 2 </P>
   <H1> 2. heading of chapter </H1>
   <H2> 2.0.1 heading of subsection </H2>
   <TABLE> table </TABLE>
   <P> explanation about table </P>
</BODY>
```

Fig. 3. Sample HTML Document

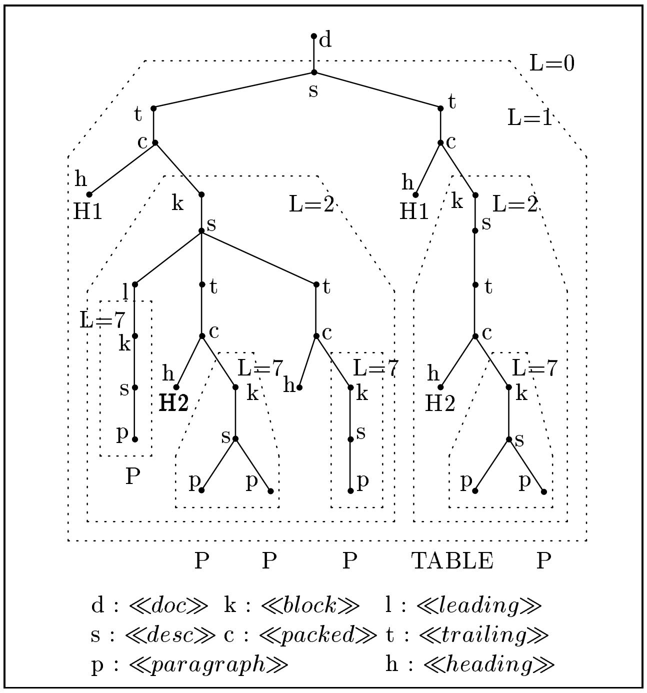

Fig. 4. Extracted Logical Tree

value given in the user profile. Finally, we construct view-pages by pruning away the subtrees whose nodes and ancestors are all marked.

4.1 Derivation of Feature Vectors

Leaf Nodes Initially, feature vectors of leaf nodes in the logical tree are derived based on the traditional tf·idf (term frequency × inverse document frequency) weighting scheme [5]. In the calculation of the idf factor, the target document set has to be fixed. We regard the set of leaf nodes as the target document set in deriving the idf. Namely, we calculate the initial tf·idf weights according to distribution of terms within the document.

Then, style-oriented tags are processed. They are usually used to emphasize some terms. Therefore, when terms occur within elements enclosed by style-oriented tags, we multiply their weights in the feature vectors depending on the types of tags. The bias factors associated with style-oriented tags are shown in Table 1.

For each leaf node n, its feature vector is formulated as follows. When a term t occurs in multiple elements enclosed by different style-oriented tags, their maximum bias factor is used:

$$v_n = (tf_t \cdot idf_{n,t} \cdot \beta_n)_{t \in \mathcal{T}}, \tag{1}$$

where β_n is the maximum bias factor associated with the style-oriented tags enclosing the term t in node n.

Non-Leaf Nodes The feature vector of each non-leaf node is given as an aggregation of feature vectors of its child nodes.

Generally, titles, headings, abstracts and introductions give good summaries of their bodies and contain many important terms [6] [7]. In the logical tree, headings appear as $\ll$*heading*$\gg$ nodes. Moreover, $\ll$*leading*$\gg$ nodes which appear as the first child node of the $\ll$*desc*$\gg$ nodes tend to play a role of abstract and introduction, as mentioned in Subsection 3.2.

To reflect this property, we calculate the feature vector of each non-leaf node n as weighted sums of those of its child nodes:

$$v_n = \sharp C \cdot \frac{\Sigma_c \alpha_c \cdot v_c}{\Sigma_c \alpha_c}, \tag{2}$$

where c is a child node of n and $\sharp C$ is the total number of the child nodes. The coefficient α_c is given in Table 2.

4.2 Generation of View-pages

To generate view-pages, we must decide substructures which are likely to be relevant to the user profile. Similarity of each node in the logical tree is calculated from its feature vector and the profile vector derived from the keyword set in the user profile. In the calculation, we have to note that the length of text (namely, the numbers of terms) associated with each node differs depending on nodes. To take this into acount, we use the C-pivot measure (see Appendix) rather than the traditional cosine measure.

We generate view-pages by pruning away subtrees whose all nodes and ancestors have similarity scores lower than the threshold value in the user profile.

tag	β_n	semantics
STRONG	5	strong emphasis
EM	3	emphasis
BIG	3	big letters
U	2	underline
B	2	bold
I	2	italic
DT	2	defined term in DL

Table 1. Bias Factors β_n Associated with Style-oriented Tags

node	α_c	node	α_c
≪*heading*≫ (H1)	15	≪*desc*≫	1
≪*heading*≫ (H2)	11	≪*leading*≫	5
≪*heading*≫ (H3)	8	≪*trailing*≫	1
≪*heading*≫ (H4)	6	≪*packed*≫	1
≪*heading*≫ (H5)	4	≪*block*≫	1
≪*heading*≫ (H6)	3	≪*paragraph*≫	1

Table 2. Coefficients α_c for Child Nodes

In this process, we have to ensure that view-pages are valid HTML documents, which the user can browse on the ordinary WWW browsers. Note that careless pruning procedure may bring about document structures which do not form valid HTML documents. For instance, an HTML document must have a BODY element and it must not be empty. Therefore, we cannot delete the BODY or its subtree even if scores of all its member nodes are less than the threshold value.

Another concern is to help the user understand the document structures through view-pages. Since the headings are important clues in understanding the whole document structures, we have decided to maintain all headings in the original documents in view-pages. In addition, some appropriate indications should be given as notes when some parts or substructures have been pruned. Moreover, simple pruning may bring about unexpected rendering of the view-page. For instance, pruning the P element away from "text <P> paragraph </P> text" brings about "text text". In this case, two "text" segments are accidentally concatenated to form a single text segment.

Based on the above consideration, in the view-page generation, we substitute a "(snip)" enclosed by a pair DIV tags for each sequence of elements which are to be pruned away. The complete procedure to generate view-pages is given as followings:

1. Calculate the similarity score of every node in the logical tree.
2. Mark the nodes whose scores are less than the threshold value. Here, ≪*doc*≫ and ≪*heading*≫ nodes are always excluded even if their scores are low, since the BODY element is mandatory and H1, ..., H6 elements are important.
3. Identify the maximal subtrees whose nodes and ancestors are all marked.
4. Replace each sequence of HTML elements associated with the subtrees with "<DIV> (snip) </DIV>".
5. Generate the output as a view-page.

Theorem *Given an arbitrary valid HTML document, the above procedure generates a valid HTML document.*

(Proof)

It is obvious from the derivation rules in Figure 2 that each sequence corresponds to the BODY element or a sequence of child elements in a BODY, DIV or

BLOCKQUOTE element. According to the HTML syntax, the BODY element cannot be replaced by a DIV element, but this case is avoided in Step 2. BODY, DIV and BLOCKQUOTE elements can contain DIV elements. Therefore, the replacement in Step 4 preserves the conformity to HTML. Thus, the above procedure derives a valid HTML document. Q.E.D.

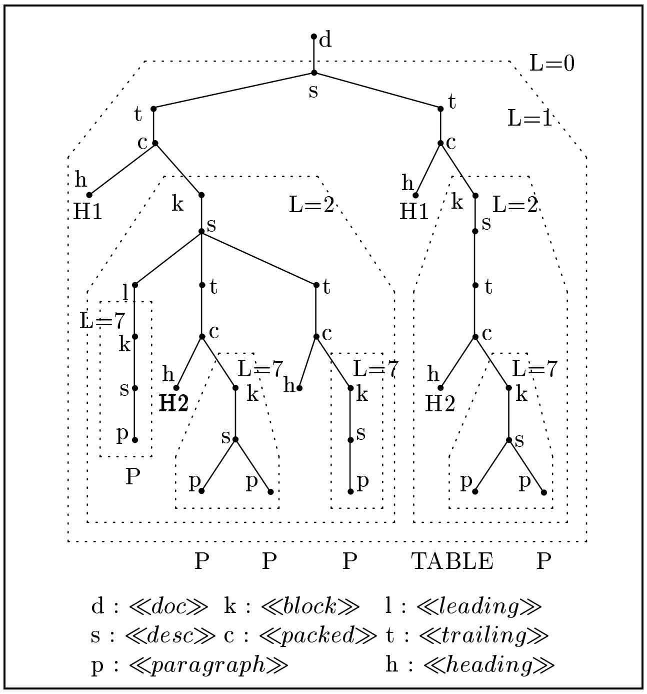

Fig. 5. Example of View-Page Generation

```
<BODY>
   <H1> 1. heading of chapter </H1>
   <DIV> (snip) </DIV>
   <H2> 1.1. heading of section </H2>
   <DIV> (snip) </DIV>
   <H2> 1.2. heading of section </H2>
   <P> body 2 </P>
   <H1> 2. heading of chapter </H1>
   <H2> 2.0.1 heading of subsection </H2>
   <TABLE> table </TABLE>
   <P> explanation about table </P>
</BODY>
```

Fig. 6. HTML Document for a View-page

Figure 5 illustrates the generation of a view-page. The circles are nodes whose scores are greater than or equal to the threshold value. The subtrees indicated by dotted lines are replaced by "`<DIV>` (snip) `</DIV>`" elements. Figure 6 shows the HTML document corresponding to the derived view-page.

5 Prototype System Architecture

In this section, we present the system architecture to implement the proposed scheme. We also show a sample session in the prototype system.

5.1 System Architecture

Overview One of criteria for our prototype system design has been to respect user-friendliness provided in the current WWW browsing environment. The prototype system architecture is shown in Figure 7. It consists of three modules.

- Browser
- Controller
- Browsing Assistance Engine

Solid arcs indicate requests and method invocations, and dotted arcs indicate responses and value returns.

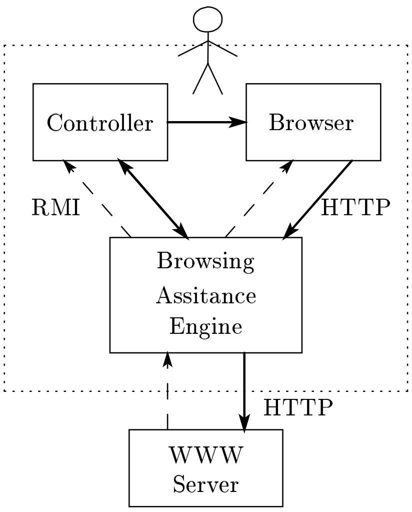

Fig. 7. Prototype System Architecture

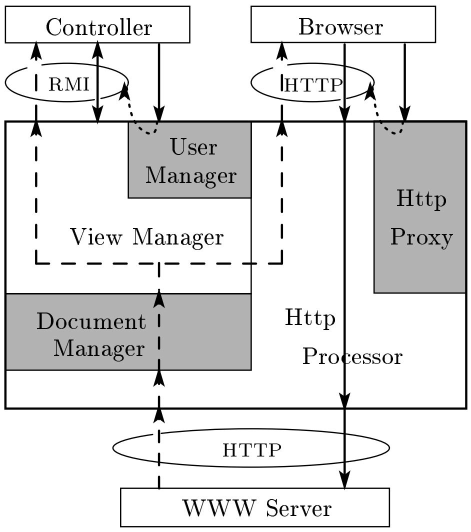

Fig. 8. Architecture of BAE

The user can use an ordinary WWW browser, such as Netscape Communicator and Internet Explorer, without any modification. The Controller and the

Browsing Assistance Engine (BAE) are original to our browsing environment. For each user, a pair of the Browser and Controller instances works as a front-end module and the BAE works as a back-end module.

The BAE is the central module in our environment, and provides the view-page for each WWW page. It works as an HTTP proxy server for the Browser, and can serve multiple pairs of the Browser and Controller instances. The BAE gets the user profile from the Controller and maintains it. It generates view-pages and their logical trees using the user profile for each user. The user can disable the view-page generation feature, when he/she wants to browse WWW pages in an ordinary manner.

The Browser displays the view-page generated by the BAE. When view-pages include hyperlinks, the user can visit other pages. Then, view-pages for the next pages will be generated and presented.

The Controller interacts with the user to get his/her user profile. When the user modifies the profile, it notifies the BAE of the modification. The Controller visually displays the logical tree of the current view-page, as described in Subsection 5.2. It helps the user to identify relevant substructures in each page.

In the prototype system, the Controller is implemented as a java applet and the BAE runs as a Java application program. The presentation on the screen given by the Browser and Controller is synchronized. The applet context is used for the synchronization.

Browsing Assistance Engine (BAE) We show the internal architecture of the BAE in Figure 8. It consists of the following five modules.

- Http Proxy
- User Manager
- Document Manager
- Http Processor
- View Manager

The first three modules are created when the BAE starts its execution. The Http Processor and View Manager are instantiated and destroyed based on demands from other modules. All the modules except the Document Manager run as threads.

The Http Proxy listens to a TCP port. When the Browser requests an HTTP resource, it instantiates an Http Processor and delegates the request to it.

The Http Processor fetches the requested HTTP resource. When the requested HTTP resource is an HTML document, the Http Processor parses it, creates the DOM object, and generates the logical tree and the view-page, by calling the Document Manager and the View Manager. When the Http Processor finishes processing the request, it is destroyed automatically.

The Document Manager is responsible for parsing documents and creating the DOM objects and logical trees. It also caches them on the memory resident hash table. URI is the hash key.

The User Manager waits for the RMI connection from a Controller instance and creates a View Manager for each connection (see below).

The View Manager communicates with the Controller to maintain the user profile and generates view-pages. View-pages are generated from the DOM objects and logical trees managed in the Document Manager. The View Manager also provides the Controller with information about the current view-page such as its URI, logical tree structure and node scores.

Typical Processing Flow

Starting-up Before invoking the Browser, we have to start the BAE, and a WWW page containing the Controller applet has to reside on the host. The user has to use the BAE as an HTTP proxy server on the WWW browser.

First, the user accesses to the WWW page containing the Controller applet. Then, the Controller is downloaded and executed in the window of the original WWW browser. It creates a new window and launches another instance of the WWW browser as the Browser. The Controller can control the Browser through the applet context. Next, the Controller interacts with the User Manager in the BAE, and its View Manager is instantiated by the User Manager.

Processing an HTTP Request After the above step, the user can browse WWW pages in an ordinary manner. When a new WWW page is requested, The Http Proxy instantiates an Http Processor to process it. If the requested resource is an HTML document, the URI is registered in the View Manager as the current URI of the Browser. Otherwise, it fetches the resource and simply returns it to the Browser.

When processing a request to an HTML document, the Http Processor checks whether it is in the cache maintained by the Document Manager. If not, it fetches and parses the the HTML document. Next, the Http Processor generates a view-page by calling the View Manager and returns it to the Browser. If the requested URI is different from the last URI, the View Manager notifies the Controller to make the presentation of the logical tree up-to-date.

Change of User Profile The user can change keywords and the threshold value in his/her user profile. In this case, the Controller notifies the View Manager. When the keywords are changed, the View Manager computes the score of each node in the logical tree again.

Next, the Controller gets the current URI from the View Manager, and makes the Browser reload the current page through the applet context. The reload request is caught by the BAE, and the View Manager generates the view-page based on the updated user profile.

Changes to the threshold value in the user profile are processed in a similar manner.

5.2 Session Example

In this subsection, we show sample usage of our browsing environment. Suppose that we want documents which are related to this paper under the W3C home page (http://www.w3.org/). Here, we select {*browsing, user, profile, interest, view, paper*} as a keywords set.

First, we turn off the view-page generation feature of the BAE, and we search candidate WWW pages querying the index server existing at the W3C site. To check highly ranked WWW pages, we turn on the view-page generation feature. Figures 9 and 10 show the Controller window and the Browser window, respectively, when we browse the first document in the list.

At the top of the Controller window, there are components to input and modify the user profile. The Controller window displays the logical tree of the current page. The color of each node is determined by its score. Nodes with the highest score are displayed in red and nodes with score 0 are in blue. Nodes with scores between the extremes are displayed in orange, yellow, green and cyan.

Nodes whose scores are greater than or equal to the threshold value are tagged with large red icons. Nodes whose scores are less than the threshold value are tagged with small cyan icons. The view-page consists of the contents corresponding to the nodes tagged with the large red icons. The user can control the threshold value interactively with the aid of the colored logical tree.

The view-page displayed in the Browser window helps our understanding the contexts. Especially, we can easily identify descriptions related to the keywords. When we want to browse the next document, we go back to the document list in the query result and select it. Then, the Browser displays its view-page. We can browse other pages similarly. When view-pages include hyperlinks, we can visit the linked pages as usual. In this way, our environment supports browsing view-pages tailored to the user's viewpoint as if they were existing WWW pages.

6 Related Works

In the proposed scheme, view-pages are generated as summaries of existing HTML documents biased by the user profile. There are several works addressing automatic summarization of documents [4] [8]. However, most of them do not take the user's interest into consideration, which is important, in particular, in the browsing process. Tombros and Sanderson proposed a method to generate summaries biased by queries [9]. However, their viewpoint focuses on the querying process, and query-biased summaries are generated to help the user select appropriate documents. Our scheme focuses on the support of the browsing process. Moreover, they do not consider the case that the targets are HTML documents. To browse summaries on ubiquitous WWW browsers, the summaries must be valid HTML documents.

When summaries are used in the browsing process, it is also important that they should be generated dynamically and automatically whenever new pages are fetched. Miike et al. developed an information retrieval system which automatically generates the abstract of each document in the query result [10]. The detailness of the abstracts can be changed interactively. However, they do not consider generation of abstracts biased by queries or user profiles.

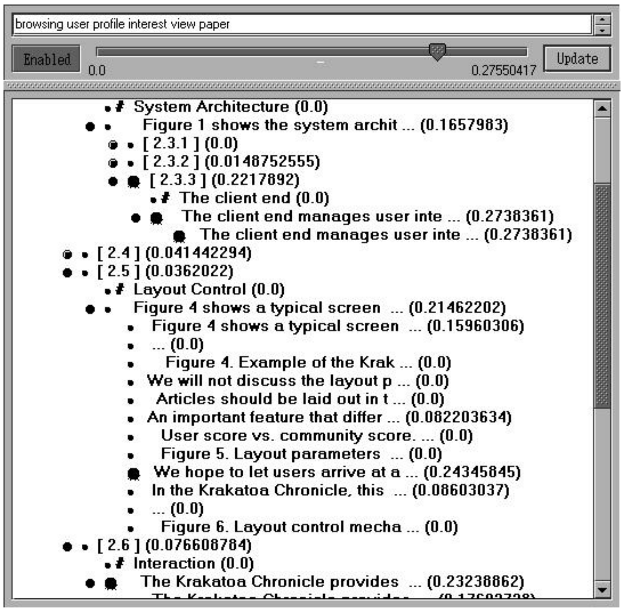

Fig. 9. Controller Window

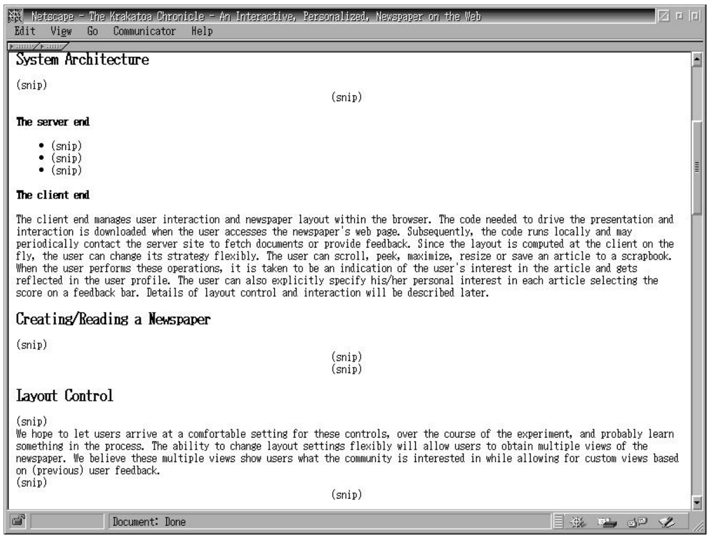

Fig. 10. Browser Window

There are a number of works to support the hyperlink navigation in the hypertext browsing. WebWatcher [3] has a feature to suggest hyperlinks which are likely to lead to the proper destinations. The system automatically acquires the user's interest from the navigation patterns. Such features may work well with our scheme.

In our scheme, scores of substructures in a document are measured. In this sense, our approach has some similarity with the passage retrieval [6] [11] [12]. However, works on the passage retrieval do not explicitly consider the browsing process. Moreover, they usually consider only simpler substructures such as sentences, paragraphs and blocks of fixed length. In our scheme, more complicated substructures are dynamically extracted in our document model.

7 Conclusion and Future Works

In this paper, we have presented a new scheme to support the WWW browsing process by dynamically providing HTML pages tailored to the user profile. This scheme makes it easy for the user to identify the interesting parts and to understand the contents of WWW pages. For this purpose, we have introduced a document model named a logical tree. In our scheme, logical document structures embedded in HTML documents are extracted in the logical tree, and summaries biased by the user profile are generated for the browsing purpose. We have also shown that the proposed scheme can be implemented non-intrusively in the current WWW browsing environment.

The remaining research issues include experimental evaluation of the proposed scheme, detailed performance analysis, and improvement of the prototype system. We are also planning to extend our scheme to make the best of hyperlinks and the referenced page contents in view-page generation. Applications of the scheme to XML documents is another interesting research issue.

References

1. P. Atzeni, A. Mendelzon and G. Mecca (eds.). *The World Wide Web and Databases*, Lecture Notes in Computer Science 1590, Springer-Verlag, 1998.
2. N. Ashish and C. A. Knoblock. Wrapper Generation for Semi-Structured Internet Sources, *ACM SIGMOD Records*, Vol.26, No.4, pp.8-15, December 1997.
3. R. Armstrong, D. Freitag, T. Joachims and T. Mitchell. WebWatcher: A Learning Apprentice for the World Wide Web, *AAAI Spring Symposium on Information Gathering from Heterogeneous, Distributed Environments*, Stanford, 1995.
4. G. Salton. *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer*, Addison-Wesley, Reading, MA, 1989.
5. G. Salton, A. Wong and C. S. Yang. A Vector Space Model for Information Retrieval, *Journal of the American Society for Information Science*, Vol.18, No.11, pp.613-620, Novemver 1975.
6. G. Salton, J. Allan and C. Buckley, Approaches to Passage Retrieval in Full Text Information Systems, *Proceedings of the 16th Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval*, pp.49-58, Pittsburgh, 1993.

7. R. Willkinson. Effective Retrieval of Structured Documents. *Proceedings of the 17th Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval*, pp.311-317, Dublin, 1994.
8. C. D. Paris. Constructing Literature Abstracts by Computer: Techniques and Prospects, *Information Processing and Management*, Vol.26, No.1, pp.171-186.
9. A. Tombros and M. Sanderson. Advantage of Query Biased Summarization in Information Retrieval, *Proceedings of the 21th Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval*, pp.2-10, Melbourne, 1998.
10. S. Miike, E. Itoh, K. Ono and K. Sumita. A Full-Text Retrieval System with a Dynamic Abstract Generation Function, *Proceedings of the 17th Annual International ACM-SIGIR Conference on Research and Development Information Retrieval*, pp.152-161, Dublin, 1994.
11. M. Kaszkiel and J. Zobel. Passage Retrieval Revisited, *Proceedings of the 20th Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval*, pp.21-29, Philadelphia, 1997.
12. J. Zobel, A. Moffat, R. Wilkinson and R. Sacks-Davis. Efficient Retrieval of Partial Documents, *Information Processing and Management*, Vol.31, No.3, pp.361-377, 1995.
13. D. K. Harman. Overview of the first TREC Text Retrieval Conference, *Proceedings of TREC Text Retrieval Conference*, pp.1-20, Washington, November 1992. National Institute of Standards Social Publication 500-207.
14. A. Singhal, C. Buckley and M. Mitra. Pivoted Document Length Normalization, *Proceedings of the 19th Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval*, pp.21-29, Zurich, August 1996.
15. D. Raggett, A. Le Hors and I. Jacobs. *HTML 4.0 Reference Specification*, 24 April 1998. http://www.w3.org/TR/REC-html40/.

Appendix: C-Pivot Measure

In the vector space model with the traditional tf·idf weighting scheme, document similarities are scored by some normalized cosine measure. In the classical cosine measure, the size of each document is not explicitly considered. Therefore, when the database contains documents of different length, smaller documents tend to have higher similarity values [13].

Singhal and others proposed the *C-pivot* measure to alleviate this problem [14]. According to the C-pivotmeasure, given a collection $\mathcal{D}$ of N documents and the vocabulary set $\mathcal{T}$, the similarity $sim(v_d, v_q)$ between the document vector v_d and the query vector v_q is calculated as follows:

$$sim(v_d, v_q) = V_d \circ V_q \tag{3}$$

$$V_d = \frac{v_d}{(1-S)\cdot U_d + S \cdot U_d} \qquad v_x = (tf_{x,t} \cdot idf_t)_{t \in \mathcal{T}} \quad (x = d, q)$$
$$V_q = \frac{v_q}{\|v_q\|} \qquad idf_t = \log(N/n_t + 1),$$

where $d \in \mathcal{D}$, U_d is the number of distinct terms in d, S is a slope constant in $[0, 1]$, $tf_{x,t}$ is the term frequency, and n_t is the number of documents including t. The slope constant is to be empirically decided.