MINING WEB PAGES FOR OBTAINING DESIGN EXAMPLES

Gongrui Zhang
Graduate School of Science and Technology
Keio University
Yokohama-shi, Kanagawa, Japan
e-mail: sosozgr@doi.ics.keio.ac.jp

Shingo Takada
Department or Laboratory
Keio University
Yokohama-shi, Kanagawa, Japan
e-mail: michigan@cs.keio.ac.jp

ABSTRACT
Designers often search for some design examples from other Web pages for reference or reuse in their design work. In order to make this possible, we first need a database that stores such Web page layouts. We propose an approach that populates a database with design examples, specifically layout examples. Our approach first crawls the Internet to retrieve Web pages, which are then mined. We especially describe how we applied the frequent subtree mining algorithm. We also show preliminary results of applying our approach.

KEY WORDS
Layout Mining, Design mining, Design Examples, Frequent Subtree Mining

1 Introduction
The user interface of any application is important as that is how users access the application. Web applications can be difficult especially when the designer lacks expertise[1, 2].

Designers often find some design examples from other Web pages for reference or reuse [2, 3, 4, 5, 6]. These design examples can give inspiration to designers [3, 4] and facilitate better design work [5, 4]. Furthermore, design examples can make the design task easier compared to starting it from scratch [7, 8]. Designers often find design examples by using search engines, such as Google. However, it is difficult to find relevant design examples with text-based keyword search [3, 6].

Designers often design the layout of the page first and then consider design details of each part in the layout [9]. A layout is an arrangement of various contents on a Web page, representing a structural design of the Web page [10].

In this paper, we especially focus on mining existing Web pages to obtain design examples that designers can refer to. Specifically, we mine the layout of Web pages based on the frequent subtree mining algorithm. The mining result will be stored in a database which designers can query to obtain relevant design examples. Our approach is useful when the designer has a specific layout in mind that he/she can specify as a query.

The contributions of this paper are as follows:

1. The concept of Box Tree and its construction for representing the structural hierarchy of visual regions in a Web page.
2. A box tree mining algorithm that is based on the frequent subtree mining algorithm.
3. The result of applying our mining approach to Web pages on the Internet.

The remainder of this paper is organized as follows. First, section 2 describes related work, Section 3 presents our proposed layout mining method, while section 4 briefly describes the implementation of our system. Section 5 then describes how our system can be used, while Section 6 shows the results of mining with our approach. Section 7 makes concluding remarks.

2 Related Work
Work concerned with extracting and reusing the layout of Web pages include [10, 11, 12, 13, 4, 14, 15, 16]. Yin, et al. [16] presented a method to extract templates from Web pages based on edit distance of the DOM. Because the structural hierarchy of the DOM does not exactly match the visual layout of the Web page, the structural information of the extracted template is different from the visual layout information of the corresponding Web page.

Bajwa, et al. [10] proposed a system called Web Layout Mining (WLM) which is based on natural language processing. WLM is useful when the user has a specific layout
already in mind. The user can use WLM to search for some samples related to his website from a database using textual keywords. Keywords describe the genre of the Web page such as News Site or Sports Site, so the relation is not always clear between the search result and the design that the designer has in mind.

Maras, et al. [11, 12] presented a semi-automatic method for extracting and reusing Web controls. They defined a control as a design element that encapsulates a certain behavior. The user can reuse an extracted control by automatically embedding it in an already existing page. Their approach can not be used to extract structural information such as the layout of a Web page.

Oh, et al. [13] proposed a system called Object-Oriented Layout Management System (OOLMS) to reuse the layout that designers created in past work. To use OOLMS efficiently, layouts need to be manually registered first, which can be time consuming.

Kumar, et al. [4, 14, 15] introduced a platform called Webszeitgeist for Web page design mining. Their work is based on design examples that designers often rely on. Webszeitgeist mines design examples with a supervised learning algorithm called Bricolage. Users can access the Webszeitgeist repository through a JSON-based design query. Our research is similar but our mining algorithm is an unsupervised learning algorithm.

3 Layout Mining

The layout of a Web page can be considered as a set of visual regions and their position relations. It is important to build a relevant model to describe the position relation and mine the layout with this model. In this paper, we will introduce a model called Box Tree to describe the position relation and mine the layout with this model.

3.1 Basic Definitions

A tree $T = (N, E, B, \text{root}(T))$ consists of a node set $N$, an edge set $E \subseteq N \times N$, a sibling relation set $B \subseteq N \times N$ and a special node $\text{root}(T)$ which corresponds to the root node of the tree. For two nodes of a tree $u$ and $v$, if there exists an edge $(u, v) \in E$, then $v$ is a child of $u$, and $u$ is the parent of $v$. The function $\text{children}(u)$ is a node set that includes all child nodes of $u$. If there exists a sibling relation $(u, v) \in B$, then $v$ is a following-sibling of $u$, and $u$ is a preceding-sibling of $v$. Tree $t$ is a unit tree of tree $T$ if and only if

1. Tree $t$ consists of only a root node and leaf nodes.
2. The leaf nodes are all the child nodes of $\text{root}(t) \in T$.

If a tree $T'$ consists of $N$ unit trees of tree $T$, we call tree $T'$ as N-pattern subtree of tree $T$. A single unit tree can be considered as a 1-pattern subtree.

Figure 2 (B) shows all of the unit trees in the tree shown in Figure 2 (A). The tree shown in Figure 2 (A) can be considered as a 4-pattern subtree if it is a subtree.

Let the database of trees be a set of trees $\{T_1, T_2, \ldots, T_n\} \in D$ where $T_i$ is a tree in database $D$. A subtree $T'$ occurs in $D$ if there exists a tree $T_i$ in $D$ such that $T'$ is a subtree of $T_i$. We define the support of a subtree $T'$ as $support(T') = \text{occur}(T') \div \text{total}(D) \times 100\%$, where $\text{occur}(T')$ is the total number of trees in $D$ that $T'$ occurs and $\text{total}(D)$ is the total number of trees in database $D$. If the support of a subtree is greater than or equal to an absolute frequency threshold, we define that subtree as a frequent subtree pattern.

3.2 Box Tree Construction

The structural information of a Web page is represented as a Document Object Model (DOM) tree. Normally, a node in a DOM tree corresponds to a visual region in the page. However, the structural hierarchy of the DOM may not exactly match the visual layout of the page [4]. So we construct a tree structure called Box Tree for representing the structural hierarchy of visual regions in the page. A node in Box Tree represents a rectangular visual region in the page. We call this node as a Box Node and the rectangular visual region as a Box. We define the coordinate origin in the page as the upper left corner. The position of each box consists of a top, right, bottom and left value. Each value is the distance between the edge of the box and the X or Y axis. The parent-child relation in the box tree corresponds to a type of position relation where the parent box is the smallest region that contains the child box. The sibling relation in the box tree corresponds to another type of position relation where the relative position of two boxes is described.

The steps of the box tree construction is shown below.

1. Remove nodes from the DOM tree that do not have any corresponding visual region in the Web page.
2. For each node in the DOM tree, do the following:
Figure 3. Example of evaluating relative position

- Calculate the position (top, bottom, left, right) of its visual region.
- Create a box node to store its position information and put it into a list called BoxNodeList.

3. For each box node A in the BoxNodeList, do the following:
   
   (a) If there exists another box node B such that B is the smallest region that contains A, we let A be a child node of B.
   
   (b) If a box node A and a box node B are contained in the same box node C, and the top position value of A is less than the top position value of B, we let A be a left-sibling node of B.
   
   (c) If a box node A and a box node B are contained in the same box node C, and the top position value of A is greater than the top position value of B, we let A be a right-sibling node of B.
   
   (d) If the top position value of A equals the top position value of B, we compare the left position value in the same manner, i.e., if the left position value of A is less than that of B, A is a left-sibling of B, and if the left position value of A is greater than that of B, A is a right-sibling of B. If the left position values are the same, we compare the bottom position values next, and if they are also the same, compare the right position values.

4. For each box node, evaluate its relative position in the parent box node.

For evaluating the relative position, we first split each box into multiple cells (N columns and M rows) and let each child in the box consist of one or more cells. Then we number each cell from 1 to N × M. Each box node that is a parent has an attribute matrix which stores the number of columns and rows that it contains. The leaf node has no attribute matrix because it has no children. Each box node that is a child has an attribute cell numbers which stores the set of cell numbers which were specified in the parent node. The root node has no attribute cell numbers because it has no parent. These two attribute values will be used for mining.

Figure 3 (A) shows a page where the parent box contains two children and the left child box has three children. The parent box in Figure 3 (A) can be split into 15 cells (5 columns and 3 rows) (Figure 3 (B)) each of which will be given numbers from 1 to 15. Its left child box can be split into 42 cells (6 columns and 7 rows) each of which will be given numbers from 1 to 42 in the same way. Figure 3 (C) shows a box tree created from Figure 3 (B). The left part of the colon in the node label represents the attribute matrix and the right part represents the attribute cell numbers. For instance, the node 6 × 7 : 7 consists of cell 7 in the parent node 5 × 3 so the value of its attribute cell numbers is 7. And it can be split into 42 cells (6 columns and 7 rows) so the value of its attribute matrix is 6 × 7. The root node in Figure 3 (C) has only attributes matrix such as 5 × 3 and the leaf node has only the attribute cell numbers such as 26,32.

3.3 Frequent Subtree Mining

The goal of frequent subtree mining is to find all frequent subtrees in a given database of trees. Apriori [1] is a well-known frequent item set mining algorithm. The idea of Apriori algorithm is to first generate candidate patterns and then evaluate the frequency of these candidate patterns. If a candidate subtree pattern appears frequently, it will be considered as a frequent pattern and be used as the base for generating new candidate patterns. The key here is how to generate new candidate patterns effectively with actual frequent patterns. There are many research on algorithms for discovering frequent subtrees based on the Apriori algorithm[17, 18, 19, 20, 21]. In these research, several
candidate generation method have been proposed but they are not valid to a practical problem because these methods are based on a simplistic labeling scheme. In this paper, our method for candidate generation is to add a unit tree to a frequent subtree pattern with the attribute matrix and cell numbers. The algorithm is as follows:

1. Find all unit trees from each tree $T$ in $D$ to form a set of unit trees $U = \{u_1, u_2, \ldots, u_n\}$. Each unit tree $u_i$ has only attribute cell numbers in leaves.

2. Create 1-pattern candidate subtrees $S_1$ from $U$. Each 1-pattern candidate subtree has only attribute matrix in root.

3. Calculate the support of each candidate subtree pattern. If the support of a candidate subtree pattern is less than a specified frequency threshold, this candidate subtree pattern will be removed.

4. The remaining candidate subtree patterns will be used to form k-pattern subtrees $S_k$. For each subtree pattern $t \in S_k$, do the following:

(a) For each leaf $l \in t$, find unit trees

$$U' = \{u'_1, u'_2, \ldots, u'_n\} \subseteq U$$

such that the attribute cell numbers of $l$ is the same as the attribute cell numbers of $\text{root}(u'_i)$.

(b) For each unit tree $u' \in U'$, let the value of attribute matrix of $l$ be the same as the value of attribute matrix of $\text{root}(u')$ and add leaves of the unit tree $u'$ to $l$ to generate a candidate (k+1)-pattern candidate subtree.

5. If (k+1)-pattern candidate subtrees are an empty set, output $S_1, S_2, \ldots, S_k$. If not, go to step 3.

Figure 4 shows the layout of three Web pages. Figure 5 shows a simple example of how frequent subtree patterns are mined from three box trees $\{T_1, T_2, T_3\} \in D$ shown in Figure 4. To illustrate this example, we assume that the specified frequency threshold is 80%. Seven unit trees (Figure 5 (A) are found from the three box trees. From the seven unit trees, five 1-pattern candidate subtrees are created first. Because the attribute matrix of the root in the
unit tree (2), (4), (7) are the same, only one 1-pattern candidate subtree is created. (The second one in the 1-pattern candidate subtrees of Figure 5 (B).) Then, the third, fourth, and fifth 1-pattern candidate subtrees are filtered from the 1-pattern candidate subtrees as 1-pattern subtrees because their support is less than the specified threshold of 80%. Thus only the first two patterns remain. The attribute cell numbers of leaves in the first 1-pattern subtree are 5 and 11 and 17, so leaves of the relevant unit trees (6), (7), (2), (3), (5) will be added to generate five 2-pattern candidate subtrees. The same procedure is applied to the second 1-pattern subtree to obtain two 2-pattern candidate subtrees. In 2-pattern candidate subtrees, only the second subtree remains after filtering as a 2-pattern subtree because its support is 100%, which is greater than 80%. Thus, the two 1-pattern subtrees and one 2-pattern subtrees are considered as the mining result.

4 Implementation

Figure 6 shows the architecture of our implementation. The crawler component crawls Web pages with some seed URLs and converts each page to a box tree. Web pages are stored in a box tree database in .mhtml format and box trees are stored in JSON format. The mining component mines the box trees and extracts frequent subtree patterns which are stored in the layout pattern database. The search engine is a server to match the layout pattern. It receives queries from the client and searches layout pattern database. The client component can be used to load the user’s source code and convert them to a box tree for query. The user can convert the entire page or just one part of a page.

The crawler component and client component is implemented as extensions of Google Chrome. The mining component is implemented with Java and the search engine component with PHP. The box tree database and the layout pattern database are implemented with MySQL.

The central function in the client component is the Selecting Layout Function, which is used when searching for design examples in our database. After finding a suitable layout within a design example, it is converted to code (HTML+CSS). This conversion is based on the selected layout’s box tree information. Each node in the box tree is converted to a div tag. The size and the position of this tag is the same as the visual region of the node.

5 Usage Example

We now show an example of how a designer can use the mined layout patterns in our layout pattern database. We assume that the designer has already coded a layout shown in Figure 7 (A) and the designer wants to refer some design examples for designing details of the layout. He/She uses the client component to load the code and selects the part that will be converted to a box tree with Selecting Layout Function. The loaded layout will be sent to the search engine for search. The box tree of the shown layout will be sent to the search engine.

The search engine finds three examples from the layout pattern database and sends them to the client (Figure 7 (B)). Here we assume the designer chooses to convert the entire page for search. The box tree of the shown layout will be sent to the search engine.

The search engine finds three examples from the layout pattern database and sends them to the client (Figure 8 (B)).
The queried layout pattern will be emphasized on each design example with red rectangles. The designer can select detailed layouts within each red rectangle using the Selecting Layout Function. We assume that the designer uses the design example in Figure 8 (A) and he/she selects the detailed layouts in the middle part in Figure 8 (D). Note that the middle part of the design example in Figure 8 (D) now has extra rectangles compared to the design example in Figure 8 (A), i.e., details of the layout of this part is now shown. The corresponding code for this layout is generated and inserted into the user’s original code automatically. The layout of the design is shown in Figure 9 (B). The designer then uses our system to search for design examples (Figure 10) corresponding to the selected layout (i.e., emphasized part) in Figure 9 (B). The result is shown in Figure 10 (A) and (B), and the designer chooses one of it (assume Figure 10 (A)). Finally, the code is updated (Figure 11 (A)). Figure 11 (B) shows the layout where the designer has selected the middle right part.

By repeatedly choosing a part of the current layout, searching for design examples, choosing one of the found example, and updating the code, the designer can quickly create the layout of a Web page. This task is made easier by showing design examples, and not just mined layout patterns.

### 6 Mining Results

<table>
<thead>
<tr>
<th>Table 1. Seed URL’s</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.alexa.com/topsites/category/Top/Arts">www.alexa.com/topsites/category/Top/Arts</a></td>
</tr>
<tr>
<td><a href="http://www.alexa.com/topsites/category/Top/Games">www.alexa.com/topsites/category/Top/Games</a></td>
</tr>
<tr>
<td><a href="http://www.alexa.com/topsites/category/Top/Health">www.alexa.com/topsites/category/Top/Health</a></td>
</tr>
<tr>
<td><a href="http://www.alexa.com/topsites/category/Top/Home">www.alexa.com/topsites/category/Top/Home</a></td>
</tr>
<tr>
<td><a href="http://www.alexa.com/topsites/category/Top/News">www.alexa.com/topsites/category/Top/News</a></td>
</tr>
<tr>
<td><a href="http://www.alexa.com/topsites/category/Top/Recreation">www.alexa.com/topsites/category/Top/Recreation</a></td>
</tr>
<tr>
<td><a href="http://www.alexa.com/topsites/category/Top/Reference">www.alexa.com/topsites/category/Top/Reference</a></td>
</tr>
<tr>
<td><a href="http://www.alexa.com/topsites/category/Top/Regional">www.alexa.com/topsites/category/Top/Regional</a></td>
</tr>
<tr>
<td><a href="http://www.alexa.com/topsites/category/Top/Science">www.alexa.com/topsites/category/Top/Science</a></td>
</tr>
<tr>
<td><a href="http://www.alexa.com/topsites/category/Top/Shopping">www.alexa.com/topsites/category/Top/Shopping</a></td>
</tr>
<tr>
<td><a href="http://www.alexa.com/topsites/category/Top/Society">www.alexa.com/topsites/category/Top/Society</a></td>
</tr>
</tbody>
</table>

We now show the results of applying our approach to mining Web pages on the Internet. We specified 15 seed URL’s (Table 1) as starting points and crawled 2000 Web pages. These 15 seed URLs were the top URL for 15 categories in the Alexa site\(^1\). To ensure the variety of Web page designs, we crawled only one page from each site. The mining result is shown in Table 2.

We used two frequency threshold values 1% and 10% to mine the 2000 Web pages. When the threshold was 1%,
Table 2. Mining Result

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of box trees</td>
<td>2000</td>
<td></td>
</tr>
<tr>
<td>Frequency threshold</td>
<td>1%</td>
<td>10%</td>
</tr>
<tr>
<td>Number of frequent patterns</td>
<td>5375</td>
<td>117</td>
</tr>
<tr>
<td>Maximum occurrence</td>
<td>1988</td>
<td>1988</td>
</tr>
<tr>
<td>Maximum K (K-pattern)</td>
<td>9</td>
<td>5</td>
</tr>
</tbody>
</table>

we found 5375 frequent patterns. The maximum value of K in K-pattern was 9 and the maximum occurrence of the frequent pattern was 1988. When the threshold was 10%, we found 117 frequent patterns. The maximum value of K in K-pattern was 5 and the maximum occurrence of the frequent pattern was 1988.

Figure 12. The pattern of maximum occurrence

We chose 1% as a frequency threshold value, because we consider that a layout pattern can be useful if it occurs in at least 20 out of the 2000 Web pages. We chose 10% as a frequency threshold value, because we want to compare differences in the number of frequent patterns between 1% and 10%. From the result, the number of frequent patterns of 1% was about 50 times as many as 10%. All patterns for 10% threshold were included in the patterns of 1% threshold.

There was only one pattern (Figure 12 (A)) that occurred 1988 times. This pattern represents a box containing another box with the same size on the Web page. This is due to the numerous existence of tags that can take only one child tag. Since nodes do not differentiate between different tags, this pattern was found to occur often. We found only one pattern where the value of K was 9 (Figure 12 (B)). Although this structure appeared more than 20 times (and thus was considered a pattern), we do not consider this pattern to be useful. This is because users will visually see only one box, i.e., it would visually be the same as the pattern in Figure 12 (A). In such a case, the possibility that users will enter a query layout that will match Figure 12(B) rather than Figure 12(A) seems to be quite low. Future work needs to confirm this hypothesis.

Figure 13. Some examples of mining result

Figure 13 shows some concrete mining result examples. In Figure 13 (A), the same layout was used in two kinds of design examples. The first one was a navigation design and the second one was a calendar design. Figure 13 (B) shows a list structure layout, which was used in part of a Web page. A wide variety of design examples were mined, suggesting the usefulness of our approach. We need to conduct an evaluation of our approach by checking if the mined patterns are actually useful.

7 Conclusion

We described an approach to support the design of a Web page layout where designers search a design database for examples. This paper especially focused on populating the database with design examples that are obtained by mining existing Web pages. The mining is based on the frequent subtree mining algorithm. We implemented our approach, and actually mined 2000 Web pages.

For future work, we first need to do an extensive evaluation to see how useful the mined patterns will be to designers. Second, we need to resolve the scalability issue. For mining a large number of Web pages, we must improve our mining algorithm. The third one is to extend our mining algorithm to mine other design attributes such as color. A more practical mining result will be obtained if we can mine several design attributes at the same time. Finally, we need to improve our algorithm for converting the layout to code. Since we can only convert to div tags, our approach...
is currently not flexible. The code should be easy to read and edit.

References


