ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.19, No.2, February 2008, pp.209−223
DOI: 10.3724/SP.J.1001.2008.00209

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

针对模板生成网页的一种数据自动抽取方法*

杨少华[1,2+], 林海略[1,2], 韩燕波[1]

[1](中国科学院 计算技术研究所 网格与服务计算研究中心,北京 100080)

[2](中国科学院 研究生院,北京 100049)

Automatic Data Extraction from Template-Generated Web Pages

YANG Shao-Hua[1,2+], LIN Hai-Lüe[1,2], HAN Yan-Bo[1]

[1](Research Center for Grid and Service Computing, Institute of Computing Technology, The Chinese Academy of Sciences, Beijing 100080, China)

[2](Graduate University, The Chinese Academy of Sciences, Beijing 100049, China)

+ Corresponding author: Phn: +86-10-62600955, Fax: +86-10-62600900, E-mail: yangshaohua@software.ict.ac.cn

Yang SH, Lin HL, Han YB. Automatic data extraction from template-generated Web pages. *Journal of Software*, 2008,19(2):209−223. http://www.jos.org.cn/1000-9825/19/209.htm

Abstract: A substantial fraction of the Web consists of pages that are dynamically generated using a common template populated with data from databases, such as product description pages on e-commerce sites. The objective of the proposed research is to automatically detect the template behind these pages and extract embedded data (e.g., product name, price…). The template detection problem is formalized and an analysis of the underlying structure of template-generated pages is made. A template detection approach is presented and the detected templates are used to extract data from instance pages. Comparing with many other existing work, the approach is applicable for both "list pages" and "detail pages". Experimental results on two large third-party test beds show that the approach can achieve high extraction accuracy.

Key words: Web; automatic data extraction; information extraction; template detection; wrapper generation

摘 要: 当前,Web 上的很多网页是动态生成的,网站根据请求从后台数据库中选取数据并嵌入到通用的模板中,例如电子商务网站的商品描述网页.研究如何从这类由模板生成的网页中检测出其背后的模板,并将嵌入的数据(例如商品名称、价格等等)自动地抽取出来.给出了模板检测问题的形式化描述,并深入分析模板产生网页的结构特征.提出了一种新颖的模板检测方法,并利用检测出的模板自动地从实例网页中抽取数据.与其他已有方法相比,该方法能够适用于"列表页面"和"详细页面"两种类型的网页.在两个第三方的测试集上进行了实验,结果表明,该方法具有很高的抽取准确率.

关键词: Web;自动数据抽取;信息抽取;模板发现;Wrapper 生成

中图法分类号: TP311 **文献标识码**: A

* Supported by the National Basic Research Program of China under Grant No.2007CB310804 (国家重点基础研究发展计划(973)); the National Natural Science Foundation of China under Grant No.60573117 (国家自然科学基金重大研究计划); the National High-Tech Research and Development Plan of China under Grant No.2006AA01A106 (国家高技术研究发展计划(863))

Received 2007-09-07; Accepted 2007-11-29

1 Introduction

Today the World-Wide Web (WWW) has become the largest distributed information repository. A substantial fraction of the Web consists of pages that are dynamically generated using a common template populated with data from databases. These types of pages constitute an important part of the so-called "Deep Web", which cannot be easily indexed by general search engines. A typical example of such a collection is the Amazon (Amazon.com. http://www.amazon.com) book pages. Moreover, the pages can be further divided into *list* and *detail pages*. A *list page* usually contains several data records (e.g., the search result page in Fig.1), while a *detail page* (e.g., the two book pages in Fig.1) contains just one record with detail information.

This paper studies the problem of automatically extracting data from template-generated web pages. Given a collection of pages generated by an unknown template, our approach detects the template and extracts embedded data without any human involvement. Table 1 shows the extracted data from detail pages in Fig.1.

Fig.1 An example list page and two detail pages from Amazon

Table 1 Extracted data from detail pages in Fig.1

Page	*A* (*$title*)	*B* (*$edition*)	*C* (*$cover*)	*D* (*$author*)	*E* (*$KeyPhrases*)	*F* (*$listprice*)	…
1	Thinking in Java	4th	Paperback	Bruce Eckel	*null*	$64.99	…
2	Thinking recursively with Java	*null*	Paperback	Eric S. Roberts	thinking recursively expression subclass …	$33.95	…
…	…	…	…	…	…	…	…

There are many challenges in automatically extracting data from template-generated web pages. Data identification is a challenging problem. In a web page, data and labels are mixed and there is no obvious way to differentiate them. For example, "by Bruce Eckel" contains a label "by" and data "Bruce Eckel". The second obvious problem is the complexity of web data schema. There are lots of repetitive, optional and nested fields in the schema of web data, and a single page may contain data from several tables of back-end databases. For example, a book may have several authors and reviews, and a review may consist of user name, publish time, content, etc. The third problem is due to the presentation-oriented characteristics of the HTML language. The same tags can be used to mark up logically different pieces of data. Even tags can also be part of data in some situations. For example, news content or blog content often contains tags like ⟨*p*⟩, ⟨*br*⟩, ⟨*img*⟩, etc.

Over the last decades, many extraction techniques have been developed, such as string and layout alignment, pattern mining, tree edit distance, statistical and machine learning methods, etc[1]. According to the experimental

results of recent works, data extraction in record level from list pages has achieved high accuracy (more than 90% in some results), while data extraction in field level from detail or list pages gets lower accuracy (about 80% average accuracy). Our approach focuses on data extraction in field level and increases the extraction accuracy rate.

This paper makes the following contributions. Firstly, we formalize the template detection problem (Section 2) and make an analysis of the underlying structure of template-generated pages (Section 4). Secondly, we propose a novel template detection approach by the organic combination the following heuristics: statistical, structural and visual layout features of the template-generated pages (Sections 5 and 6). Thirdly, experimental results on two thirty-party test beds show that our approach can achieve high extraction accuracy in both list and detail pages (Section 7).

2 Problem Definition

2.1 Representation of Web pages

Appropriate representation of web pages can bring convenience to the extraction task. In the paper, three types of tokens: *StartTagToken*, *EndTagToken* and *TextToken* are introduced to represent the building elements of web pages and two data structures: *token sequence* and *token tree* are used to represent the structure aspect (see Fig.2).

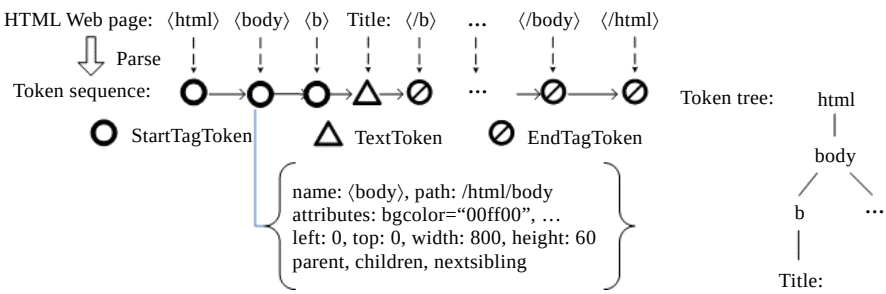


Fig.2 Representation of web pages: Token sequence and token tree

A *StartTagToken* represents a HTML start tag, such as 〈*body*〉, 〈*table*〉, etc. It contains some basic properties (tag *name*, *path* and *attributes* in the DOM tree), visual layout properties (*left*, *top*, *width* and *height*) and token pointers to its parent, children and next sibling. Token pointers can be used to construct token tree. An *EndTagToken* represents a HTML end tag, e.g., 〈/*body*〉. It contains properties like *name*, *path*, etc. Text between HTML tags is represented by one or more than one *TextToken*, which has similar properties to *EndTagToken*. What's different is that the name of *TextToken* represents the text content and the path is like "…/*b*/*text*()".

A *token sequence* is a linear data structure that is made up of all these three kinds of tokens. On the other hand, a *token tree* only contains *StartTagTokens* and *TextTokens*, since *EndTagTokens* are unnecessary to reserve the token hierarchy of HTML documents. For every web page, there is a root token tree, while every offspring node of the root token also forms a token tree itself. Comparing with simple string or tag-tree, our representation of web pages reserves more information of HTML documents and includes visual information.

2.2 Template and template detection

Definition 2.1 (**template**). A *template T* of a set of web pages is a regular expression over an alphabet $\Sigma=M\cup L\cup D$ and a set of operators (•,*,?), where:

- *M* is the set of template tags, including both the HTML start tags and end tags. Since a common HTML tag can be used as different elements of *M*, we use a HTML tag plus a suffix instead, e.g. div_1. In some cases, if

HTML tags are part of data, these tags should not be classified to *M*.

- *L* is the set of labels that appear in web pages. A label is a text string that is not part of data or a template tag. For example, in Fig.3, text strings like "by", "Reviewer" and "Rate" are labels. Intuitively, labels are those text strings shared by a majority of web pages.

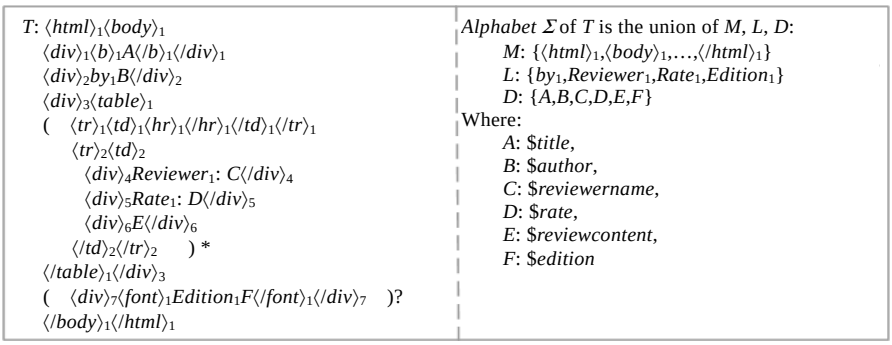


Fig.3 An example template

- *D* is the set of data fields. In most situations, a data field corresponds to a column of a table (view) in a relational database. In this paper we use upper letters such as *A*, *B* to represent anonymous fields, while known fields are represented by '$' plus the field names (e.g., $*title*, $*price*).

Since the original database field names are generally not encoded in Web pages, we use the anonymous fields in the template. Discovering data field names is a separate research problem called *attribute labeling* which is beyond the scope of this paper.

Example 2.1. Figure 4 shows an example which contains threes pages generated by the template in Fig.3. Example pages are given as the form of token trees. Each token has a name plus a suffix to differentiate with others.

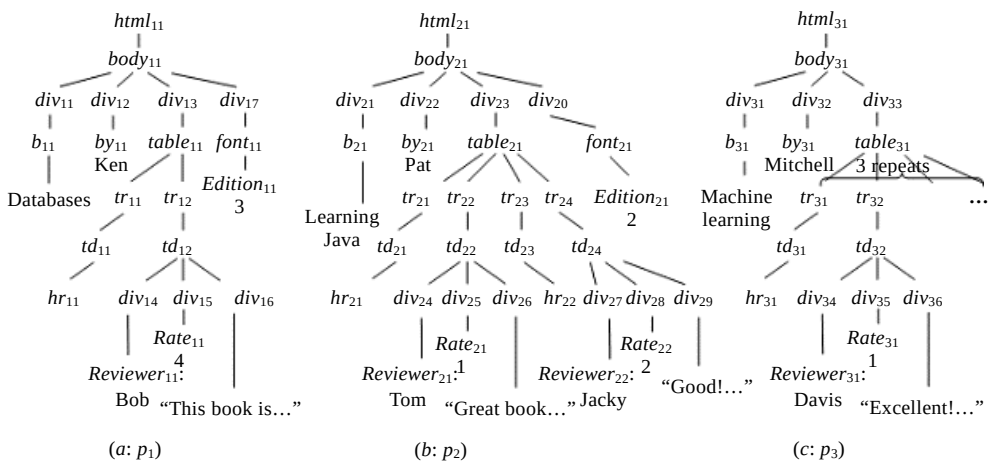


Fig.4 Three example pages generated by the template in Fig.3

Definition 2.2 (**template detection**). Given a set of web pages generated by some unknown template *T*, *template detection* is the process of deducing the template *T*.

With the help of these definitions, we can see that filling the data fields in a template with appropriate data creates a page, while template detection can be seen as the reverse process of page creation. Formally, the template detection problem can be seen as an instance of regular grammar inference problem from positive examples, which is a well-known and extensively studied problem. However, it has been proven in theory that the correctness of deducing regular grammars from positive examples alone is undecidable[2]. Fortunately, for template detection,

specific heuristics can be used to enhance the task, such as statistical, structural and visual layout features of template-generated pages. This paper proposes a novel template detection approach using these heuristics.

3 Overview of Our Approach

Given a set of example pages, our approach can be briefly described as follows: *Firstly*, we group the tokens in the pages into CTokens, where a ctoken denotes the set of tokens generated by the same element of an alphabet. For example, $\{html_{11},html_{21},html_{31}\}$, $\{by_{11},by_{21},by_{31}\}$ and $\{Ken,Pat,Mitchell\}$ are three ctokens corresponding to elements $\langle html\rangle_1$, by_1 and B ($author) in Example 2.1. This step can be seen as the process of finding "letters" of the unknown template's alphabet. Note that tokens in the same ctoken may distributed in the same page (generated by while iteration) as well as across different pages (generated by applying the same template multi-times). *Secondly*, we construct the target template by analyzing the occurrence patterns of the ctokens. This step can be seen as the process of recovering the regular expression over the alphabet.

Figure 5 shows the following main steps of our approach and each step's input and output.

- Tokenization. In this step, the given web pages are transformed into token sequences (trees). We use a word-level tokenization mechanism, which means that any text string between two tags is parsed into one or more TextTokens. For example, text string "Rate 4" in p_1 is parsed into two separate TextTokens: $Rate_{11}$ and "4". However, some special text strings that match pre-defined patterns (regular expressions) will not be splitted. At the writing time of this paper, we predefine the following patterns: date, time and link. Of course, other well-known patterns can be pre-defined to help tokenization. For StartTagTokens, visual layout properties (left, top, width, height) are computed by Mozilla/Gecko (http://www.mozilla.org/newlayout/), which is a well-known web browser layout engine.
- Compute ctokens. In this step, tokens are clustered into ctokens using various heuristics, such as token properties (e.g. name, path), the structural context and statistical information, etc. The resultant set of the computed ctokens will be used as the alphabet of the unknown template. More accurate definition of ctoken and other core concepts and their characteristics are described in Section 4. Section 5 discusses the algorithm for computing ctokens in detail.
- Construct the template and extract data. Based on the ctokens computed in Step 2, another algorithm is developed to construct the target template by analyzing the occurrence patterns of the ctokens in the token sequences. Section 6 presents the detail of the algorithm. When the template is constructed, we can construct a nondeterministic finite automaton (NFA) from the template, and then use the NFA to extract data from instance pages. Since this process is straightforward, the paper does not describe this topic.

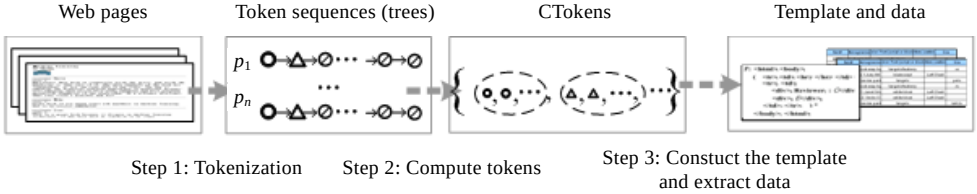


Fig.5 Overview of our approach

4 CTokens

This section defines the core concepts that clarify the underlying structure of template-generated Web pages and presents their characteristics, which serve as the rational behind the design of our template detection algorithm. Note that most of the characteristics are based on our observations and hold in majority cases, so they can be

leveraged as powerful heuristics. However, they are NOT always true. The concepts below are in the context of a set of token sequences, which are tokenized from example web pages generated by an unknown template *T*. *T* consists of an alphabet Σ and a regular expression over the alphabet.

Definition 4.1 (**CToken**). A *ctoken* is a maximal set of tokens generated by the same element of Σ. If the element is from $M(L,D)$, the ctoken is also called a *markup-ctoken* (resp. *label-ctoken*, *data-ctoken*). The tokens in a ctoken are also called as the *appearances* of the ctoken in the web pages.

Note the one to one correspondence between a ctoken and an element of Σ, we can use the name of the corresponding element to denote the ctoken, e.g. $\langle html\rangle_1$ is used to denote the ctoken $\{html_{11},html_{21},html_{31}\}$.

From the definition of ctoken, we have the following characteristics:

- *C*1: Tokens have the same name if they are in a markup-ctoken or label-ctoken. For example, tokens generated by $\langle html\rangle_1$ have the same name "$\langle html\rangle$". However, tokens in data-ctokens usually have different names, e.g., tokens generated by *B*(\$*author*) have names: "Ken", "Pat" and "MitChell".
- *C*2: Tokens in the same ctoken have the same path. This characteristic can be easily verified in example 2.1. For example, tokens generated by $\langle tr\rangle_1$ have the same path "/*html*/*body*/*div*/*table*/*tr*". However, the contrary is not true in general. For example, tr_{11} and tr_{21} have the same path, but they belong to different ctokens.
- *C*3: If two StartTagTokens belong to the same ctoken, their corresponding EndTagTokens must be in the same ctoken (and vice-versa). For example, $html_{11}$ and $html_{21}$ belong to $\langle html\rangle_1$, their corresponding EndTagTokens $\langle /html\rangle_{11}$ and $\langle /html\rangle_{21}$ must belong to the same ctoken ($\langle /html\rangle_1$).

The characteristics *C*1 and *C*2 above can be used to group the tokens initially and the characteristic *C*3 can be used to construct a new ctoken using the corresponding EndTagTokens of the set of StartTagTokens.

Definition 4.2 (**equivalence forest**). An *equivalence forest* is a maximal set of token trees whose root tokens belong to the same markup-ctoken. The markup-ctoken is also called as the *root ctoken* of the equivalence forest.

For example, $\{T\langle html_{11}\rangle,T\langle html_{21}\rangle,T\langle html_{31}\rangle\}$, $\{T\langle div_{17}\rangle,T\langle div_{20}\rangle\}$, $\{T\langle tr_{11}\rangle,T\langle tr_{21}\rangle,T\langle tr_{23}\rangle,T\langle tr_{31}\rangle,T\langle tr_{33}\rangle,T\langle tr_{35}\rangle\}$ are three of the 17 equivalence forests in Example 2.1. We use $T\langle tokenid\rangle$ to denote the token tree rooted at the token identified by *tokenid*. Intuitively, an equivalence forest is used to denote fragments of web pages generated by the same fragment of a template. Initially, the root token trees of example pages make up a bootstrap equivalence forest. Then, by examining the internal structure of these token trees, more fine grained equivalence forests can be discovered.

Definition 4.3 (**skeleton CToken**). A ctoken *c* is a *skeleton ctoken* with respect to an equivalence forest *F* iff there is a one-to-one mapping *m* between the tokens in *c* and the token trees of *F*, where $m(t)=T\langle r\rangle$ means that token *t* occurs in token tree $T\langle r\rangle$.

Obviously, the root ctoken of an equivalence forest is a skeleton ctoken, since every token tree contains exactly one root token. Because of the one-to-one mapping between StartTagTokens and EndTagTokens, if *c* is a skeleton ctoken which consists of StartTagTokens, a ctoken c' consisting of the corresponding EndTagTokens is also seen as a skeleton ctoken, though EndTagTokens do not occur in the token trees. Take an equivalence forest $F=\{T\langle tr_{11}\rangle,T\langle tr_{21}\rangle,T\langle tr_{23}\rangle,T\langle tr_{31}\rangle,T\langle tr_{33}\rangle,T\langle tr_{35}\rangle\}$ for example, *F* contains six skeleton ctokens: $\langle tr\rangle_1$, $\langle /tr\rangle_1$, $\langle td\rangle_1$, $\langle /td\rangle_1$, $\langle hr\rangle_1$, $\langle /hr\rangle_1$.

From the definition of skeleton ctoken, we have the following characteristics:

- *C*4: Skeleton ctokens to the same equivalence forest *F* have the same number of tokens, which is equal to the number of token trees in *F*.
- *C*5: For every two skeleton ctokens c_1 and c_2 to the same equivalence forest *F*, their appearances in the

same token tree have a fixed occurrence order in the token sequences. Let $T\langle r\rangle$ be any one token tree of F, $t_1 \in c_1$ and $t_2 \in c_2$ are two tokens in $T\langle r\rangle$, such that t_1 is always before (or after) t_2. The characteristic makes sure that for the skeleton ctokens to an equivalence forest, their appearances occurring in the same token tree have a fixed occurrence order in the token sequences.

The characteristic *C*5 is also called as the "*ordered*" characteristic of skeleton ctokens. Given a set of skeleton slots, we can sort these skeleton slots by the occurrence order of their appearances in the token sequences. Based on the two characteristics above, we define another concept "*skeleton slot*" below and then derive the "*nested*" characteristic of skeleton ctokens.

Definition 4.4 (**skeleton slot**). Given m ordered skeleton ctokens $c_1,c_2,\ldots,c_m$ with respect to an equivalence forest F that is made up of k token trees, a *skeleton slot* between two adjacent skeleton ctokens c_l and c_{l+1} ($1\le l\le m-1$) is the union set of A_i ($1\le i\le k$), where A_i is a set of tokens that occur between their appearances in the ith token tree of F. We use 2-tuple $\langle c_1,c_2\rangle$ to represent a skeleton slot between two skeleton ctokens c_1 and c_2.

Fig.6 shows skeleton slots in the equivalence forest $\{T\langle html_{11}\rangle,T\langle html_{21}\rangle,T\langle html_{31}\rangle\}$. For example, the skeleton slot $\langle\langle body\rangle_1,\langle b\rangle_1\rangle$ has three tokens: $div_{11},div_{21},div_{31}$, which can be computed as follows: *Firstly*, for each token tree, the appearances of the two skeleton ctokens are found and form a token pair (e.g. $body_{11}$ and b_{11} in $T\langle html_{11}\rangle$). *Secondly*, tokens between these token pairs in the token sequences are found and added into the skeleton slot.

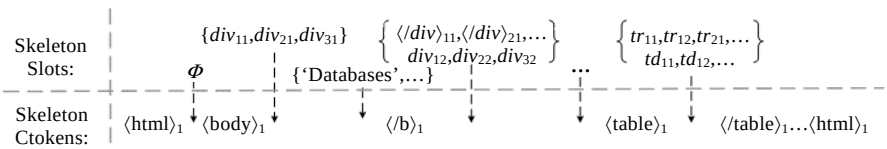


Fig.6 Skeleton slots in an equivalence forest

From the above definitions, we derive the "*nested*" characteristic of skeleton ctokens as follows:

- *C*6: Given m ordered skeleton ctokens $c_1,c_2,\ldots,c_m$ with respect to an equivalence forest, we can build $m-1$ skeleton slots: $s_1,s_2,\ldots,s_{m-1}$. If two tokens $t_1 \in s_k$ and $t_2 \in s_l$ ($k\ne l,1\le k,l\le m-1$), then t_1 and t_2 belong to different ctokens. For example, in Fig.6, $div_{11} \in \langle\langle body\rangle_1,\langle b\rangle_1\rangle$ and $div_{12} \in \langle\langle /b\rangle_1,by_1\rangle$, such that they can not belong to the same ctoken. Actually, div_{11} belongs to $\langle div\rangle_1$, while div_{12} belongs to $\langle div\rangle_2$.

Intuitively, skeleton ctokens are used to denote partition points that divide the tokens inside a token tree into disjoint sets. Armed with the characteristics *C*5 and *C*6, these two concepts provide us a powerful tool for applying *divide and conquer strategy* on discovering ctokens and templates.

5 Computing CTokens

This section discusses the algorithm for computing ctokens. We will describe how to use the characteristics presented in Section 4 in our algorithm. In particular, two key techniques of the algorithm are introduced. The first technique is on how to compute skeleton ctokens in an equivalence forest. The other technique is how to compute equivalence forests in a skeleton slot. Ctokens computed by the algorithm are *possible* ctokens because the algorithm itself can't ensure whether they are the *real* ones. In this paper, we still use ctoken to refer to *possible* ctoken and the correct sense can be identified by the context around the word.

5.1 The computation algorithm

Given a collection of token sequences (trees) that represent a collection of example web pages, the ctokens are computed as follows:

Algorithm 1. *Compute_CTokens*(*P*).

Input: A collection of token trees $P=p_1,p_2,\ldots,p_n$;

Output: The collection C of ctokens.

L1: $C\leftarrow\Phi$, $Q\leftarrow$new *Queue*()

L2: $Q.enqueue(\{T\langle html_{11}\rangle,T\langle html_{21}\rangle,\ldots,T\langle html_{n1}\rangle\})$

L3: **while** Q is not empty **do**

L4: $F\leftarrow Q.dequeue()$

L5: $C\leftarrow C\cup Compute\_Skeleton\_CTokens(F)$

L6: **for** each skeleton slot $\langle c_m,c_n\rangle$ in F **do**

L7: $(E,D)\leftarrow Compute\_Equivalence\_Forests(\langle c_m,c_n\rangle)$

L8: $C\leftarrow C\cup D$ /*D is a collection of data-ctokens and label-ctokens*/

L9: **for** each equivalence forest f in E **do**

L10: $Q.enqueue(f)$

L11: **end for**

L12: **end for**

L13: **end while**

At L2 of Algorithm 1, we assume that the input collection of token trees ($\{T\langle html_{11}\rangle,T\langle html_{21}\rangle,\ldots,T\langle html_{n1}\rangle\}$) is an equivalence forest and use it as the bootstrap equivalence forest. This assumption is held in almost all HTML based web pages.

The object Q is a FIFO queue of equivalence forests. For each iteration (from L4 to L12), firstly, the head equivalence forest of Q is removed and denoted by F; secondly, skeleton ctokens in F are computed; thirdly, for each skeleton slot with respect to F, more equivalence forests are computed and inserted into Q. The third step makes the iteration continually and is guided by the characteristic C5 and C6 described in Section 4. If the computed skeleton slots of are correct (which means that the computed skeleton ctokens are the *real* ones), these characteristics ensure that for each ctoken c and each computed skeleton slot s, $c\subseteq s$ or $c\cap s=\varnothing$.

Algorithm 1 has two key procedures: firstly, computing skeleton ctokens in an equivalence forest; and secondly computing equivalence forests and data-ctokens in a skeleton slot. The detail is given in the following subsections.

5.2 Computing skeleton CTokens

Given an equivalence forest $F=\{T\langle r_1\rangle,T\langle r_2\rangle,\ldots,T\langle r_k\rangle\}$, the procedure of computing skeleton ctokens is outlined as follows:

1. $S\leftarrow\Phi$, S is used to store skeleton ctokens.
2. Cluster tokens that occur in F with the same path and name into possible ctokens.
3. For each possible ctoken c, if there is a one-to-one mapping between the tokens in c and the token trees of F, add c to S.
4. For each possible ctoken $c\in S$ and made up of StartTagTokens, create a new ctoken c' that consists of the corresponding EndTagTokens of the StartTagTokens in c and add c' into S.
5. Make sure skeleton ctokens in S satisfy the "*ordered*" characteristic ($C5$).

The 2nd step of the procedure is based on the characteristics $C1$ and $C2$. The 3rd step uses the definition of skeleton ctoken. In the 4th step, a new ctoken is created based on the characteristic $C3$. In the last step, some computed ctokens that can't satisfy the characteristic $C5$ will be removed from S.

Take the equivalence forest $\{T\langle html_{11}\rangle,T\langle html_{21}\rangle,T\langle html_{31}\rangle\}$ for example, 9 skeleton ctokens are computed in this procedure: $\langle html\rangle_1$, $\langle body\rangle_1$, $\langle b\rangle_1$, $\langle /b\rangle_1$, by_1, $\langle table\rangle_1$, $\langle /table\rangle_1$, $\langle /body\rangle_1$, $\langle /html\rangle_1$. Note that some skeleton ctokens

can't be found in this procedure, such as skeleton data-ctokens. These skeleton ctokens will be found in subsequent procedures.

5.3 Computing equivalence forests

This procedure computes the equivalence forests as well as data-ctokens and label-ctokens in a given skeleton slot. The equivalence forest computation problem is a special clustering problem, where token trees rooted at tokens that belong to the same markup-ctoken are grouped into an equivalence forest. We propose a clustering algorithm for this specific problem, using the characteristics presented in Section 4 and an additional observation: *token trees in an equivalence forest usually have similar structural and visual layout features*. We define a measure of the similarity between two token trees and use the average linkage clustering method to group similar token trees.

Given a skeleton slot $\langle c_m,c_n\rangle$ with respect to an equivalence forest F, the procedure of computing equivalence forests is outlined as follows:

1. $T\leftarrow$the collection of most "top" token trees rooted at StartTagTokens in the skeleton slot.
2. Partition T into $\{T_1,T_2,\ldots,T_k\}$, $T_i\subseteq T$ $(1\le i\le k)$ and $\bigcup_{i=1}^{k}T_i=T$ by the following rule: If two token tree roots have the same path and name, then both token trees belong to the same subset of T.
3. For each T_i $(1\le i\le k)$ rooted at StartTagTokens, cluster the token trees in T_i into groups based on their similarities. Each group is an equivalence forest.
4. Compute data-ctokens and label-ctokens from the "top" TextTokens in the skeleton slot.

In the 1st step, a most "top" token tree is a token tree that satisfies: let r be the root of the token tree, ① $r\in\langle c_m,c_n\rangle$; ② the parent of r is not in $\langle c_m,c_n\rangle$. The 2nd step is guided by the characteristics C1 and C2. In the 3rd step, we use the average linkage clustering method to group similar token trees into equivalence forests and the clustering process is guarded by a similarity threshold. The 4th step first cluster the TextTokens with the same name into label-ctokens, and then the remaining TextTokens are grouped into data-ctokens.

Take the skeleton slot $\langle\langle table\rangle_1,\langle/table\rangle_1\rangle$ for example. The result of the 2nd step is a set made up of 12 tokens trees: $T\langle tr_{11}\rangle,T\langle tr_{12}\rangle,T\langle tr_{21}\rangle,\ldots$. Then after the clustering process in the 3rd step, we gain two equivalence forests: $F_1=\{T\langle tr_{11}\rangle,T\langle tr_{21}\rangle,T\langle tr_{23}\rangle,T\langle tr_{31}\rangle,T\langle tr_{33}\rangle,T\langle tr_{35}\rangle\}$ and $F_2=\{T\langle tr_{12}\rangle,T\langle tr_{22}\rangle,T\langle tr_{24}\rangle,T\langle tr_{32}\rangle,T\langle tr_{34}\rangle,T\langle tr_{36}\rangle\}$.

The similarity of two token trees t_1 and t_2, denoted by $s(t_1,t_2)$, is defined as:

$$s(t_1,t_2)=\frac{x_1^T\bullet x_2}{\|x_1\|\|x_2\|},$$

where: x_1 and x_2 are feature vectors of t_1 and t_2, $\|x_1\|\|x_2\|=(x_1^Tx_1x_2^Tx_2)^{1/2}$.

This definition is called as *Cosine Similarity* and can be geometrically interpreted as the cosine of the angle between x_1 and x_2. We use the binary valued features, which means if a token tree possesses the ith feature then the value of the ith vector component is 1 (otherwise, 0). The vector product $x_1^T\bullet x_2$ is the number of features possessed by both t_1 and t_2, and $\|x_1\|\|x_2\|$ is the geometric mean of the number of features possessed by x_1 and the number possessed by x_2. Thus, the value of $s(t_1,t_2)$ is between 0 and 1.

For the technical convenience, every feature is identified by a string called *key* and two features are the same if they have the same key. We only choose those structural and visual layout features of the token trees to compute the similarity. At the writing time of this paper, we propose the following types of features (using the subtrees t_1, t_2, t_3 in Fig.7 for example):

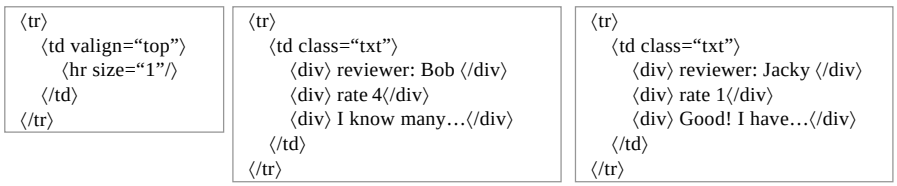
1) *Tag Feature*. Every StartTagToken in a token tree is mapped into a *tag feature* whose key is the token's path, e.g. "/*html*/*body*/*div*/*table*/*tr*". Two StartTagTokens in different token trees mapped into the same feature are also called *common tags* of the token trees.

2) *Attribute Feature*. The attributes of the common tags are mapped into *attribute features* whose key is like "…/*tr*/*td*@*valign*=“top””, where “valign” and “top” are the attribute’s name and value.

3) *Width* (*Height*) *Feature*. The widths of the common tags are mapped into *width features* whose key is like "…/*tr*/*td*#*width*=800", where “800” is the width of the common tag ⟨*td*⟩. A *height feature* is defined as similar to a width feature.

4) *Text Feature*. Every TextToken in a token tree is mapped into a *text feature* whose key is the token’s path, e.g., “rate” in t_2 is mapped into a text feature with a key "…/*tr*/*td*/*div*/*text*()".

5) *Specific Common Text Feature*. If two token trees both have text strings in the same path, we compare these strings and construct *specific common text features* if the strings have the same characteristics. For example, if two strings are equal in content, we construct a "*same content*" feature whose key is like "…/*div*/*text*()#*content*=$*text*". Besides content, other characteristics such as length, pre-defined patterns are considered as well. In our prototype system, we predefine the following patterns: numbers, date, time, price and link. For example, if two strings both contain numbers, a common feature is constructed with a key like "…/*div*/*text*()#*hasnumbers*".



(*a*: t_1) (*b*: t_2) (*c*: t_3)

Fig.7 HTML fragments of three token trees in a skeleton slot

Table 2 shows the features of t_1 and t_2 (we use the tag name as a substitute for the path in the feature key), and the similarity of t_1 and t_2 can be computed by $s(t_1,t_2)=3/\sqrt{8\times 9}=0.35$.

Table 2 Features of t_1 and t_2 in Fig.7

Feature	t_1	t_2	Feature	t_1	t_2	Feature	t_1	t_2	Feature	t_1	t_2
⟨*tr*⟩	**1**	**1**	⟨*td*⟩@*valign*=“*top*”	1	0	⟨*tr*⟩#*height*=35	0	1	⟨*td*⟩#*height*=35	0	1
⟨*td*⟩	**1**	**1**	⟨*td*⟩@*class*=“*txt*”	0	1	⟨*td*⟩#*width*=796	1	0	⟨*div*⟩/*text*()	0	1
⟨*hr*⟩	1	0	⟨*tr*⟩#*width*=800	**1**	**1**	⟨*td*⟩#*width*=780	0	1			
⟨*div*⟩	0	1	⟨*tr*⟩#*height*=2	1	0	⟨*td*⟩#*height*=1	1	0			

Given a collection of token trees and a similarity threshold δ, we use the average linkage clustering method to group similar token trees into equivalence forests. Initially, every token tree is in its own cluster. Then the method repeats the following two steps: ① calculate the similarity between every two clusters using average values of similarities between every member in a cluster and all other members in another cluster; ② find the two clusters with the highest average similarity $s_{\max}$, if $s_{\max}\geq\delta$ then the two clusters are joined together to form a new cluster, otherwise exit the repeating process. Finally, the result clusters are the equivalence forests.

6 Constructing Templates

This section discusses the algorithm of template construction over the ctokens discovered in Section 5. From the process of computing equivalence forests, we can conclude that equivalence forest also has the "*nested*" characteristic: Equivalence forests are always "*nested*" in the same skeleton slot. Thus, we can apply *divide and conquer strategy* to our template constructing algorithm. The algorithm is described as follows in detail:

Algorithm 2. *Construct_Template*(*F*,*C*).

Input: An Equivalence Forest $F=\{T\langle r_1\rangle,T\langle r_2\rangle,\ldots,T\langle r_n\rangle\}$, a collection C of ctokens;

Output: The template fragment (regular expression) of F.

L1: $R\leftarrow$Empty regular expression over C

L2: $c_1,c_2,\ldots,c_m\leftarrow$Find all the skeleton ctokens in C with respect to F and sort them by their appearances.

L3: **for each** skeleton slot $\langle c_i,c_{i+1}\rangle$ $(1\le i\le m-1)$ **do**

L4: $R\leftarrow R\bullet c_i$ /*"$\bullet$" is the concatenation operator.*/

L5: $O\leftarrow$Find those ctokens in C which consists of the most "top" tokens in $\langle c_i,c_{i+1}\rangle$.

L6: $Kleene\leftarrow\Phi$, $Optional\leftarrow\Phi$, $Alternation\leftarrow\Phi$ /*used to store regular expressions over C*/

L7: **for** each ctoken $o\in O$ **do**

L8: if there exist at least one token tree of F that o has more than one appearance, then add (o)* to *Kleene*, otherwise, add (*o*)? into *Optional*.

L9: **end for**

L10: Merge the regular expressions in *Kleene* and *Optional* respectively. The alternation expressions merged from optional expressions are added into *Alternation*.

L11: $Exprs\leftarrow Kleene\cup Optional\cup Alternation$.

L12: Sort the regular expressions in *Exprs* by the occurrence order of tokens generated by them.

L13: **for each** regular expression $expr\in Exprs$ **do**

L14: Every ctoken c that consists of StartTagTokens in $expr$ is replaced by the regular expression computed by *Construct_Template*(f,C) recursively, where f is the equivalence forest using c as its root ctoken.

L15: $R\leftarrow R\bullet expr$

L16: **end for**

L17: **end for**

L18: $R\leftarrow R\bullet c_m$

L19: **return** R

Algorithm 2 is designed as a recursive procedure. Skeleton ctokens are concatenated to form the "skeleton" of the target regular expression. However, it is much more complicated to deduce regular expressions from skeleton slots (from L3 to L17). At L5, a most "top" token in $\langle c_i,c_{i+1}\rangle$ is a token whose parent is out of $\langle c_i,c_{i+1}\rangle$, for these tokens are usually located at the most top of the token tree comparing with other tokens. At L10, merging the regular expressions in *Kleene* and *Optional* is the most difficult subprocedure in the whole algorithm. For convenience, fragments of tokens generated by a regular expression e are also called as the *appearances* of e. The merging procedure is described as follows:

For the set *Kleene*, the algorithm repeats looking for two expressions (r)* and (s)* in *Kleene* that satisfy any one of the following conditions until none can be found:

1) If $\forall T\in F$, suppose T contains k appearances of r (denoted by $r_1,r_2,\ldots,r_k$) and l appearances of s (denoted by $s_1,s_2,\ldots,s_l$), such that $k=l$ and r_i is always directly followed by s_i $(1\le i\le k)$ in the token sequence of T, then, merge (r)* and (s)* into ($r\bullet s$)*.
2) If $\forall T\in F$, every appearance of s always follows some appearance of r directly, and $\exists T\in F$, T contains some appearance of r which is not followed by an appearance of s directly, e.g. "$rs\ldots\underline{r}\ldots rs\ldots$", then, merge ($r$)* and ($s$)* into ($r\bullet s$?)*.
3) If $\forall T\in F$, every appearance of r is always followed by some appearance of s directly, and $\exists T\in F$, T contains some appearance of s which does not follow an appearance of r directly, e.g. "$rs\ldots\underline{s}\ldots rs\ldots$",

then, merge (r)* and (s)* into (r?•s)*.

For the set Optional, the algorithm repeats looking for two expressions (r)? and (s)? in Optional that satisfy any one of the following conditions until none can be found:

1) If $\forall T \in F$, r and s have the same number (0 or 1) of appearances in T, and "$r_i s_i$" is part of token sequence of T when the number is 1 (r_i denotes the appearance of r, and so is s_i), then, merge (r)? and (s)? into (r•s)?.
2) If $E(r) \cap E(s) = \varnothing$ and $E(r) \cup E(s) = F$, where $E(expr)$ is the set of trees in F which contain appearances of $expr$, then, merge (r)? and (s)? into (r|s), which is an alternation expression.

Essentially, constructing templates over ctokens is an instance of regular grammar inference problem from positive examples and the correctness is undecidable in theory. Thus, the algorithm may be invalid in some case. However, empirical results on two large third-party datasets show that the algorithm works well in most cases.

7 Experiments

In this section, we present the empirical results on two third-party datasets using our approach and compare the results with other three mostly-referenced methods. The results show that our approach can achieve high extraction accuracy in both list and detail pages. At the last, we analyze the results and explore the reasons behind.

7.1 Experimental setup

Based on the approach described above, we have built a prototype of a wrapper generation system called **Grubber**. The system is implemented as an extension of Mozilla Firefox and mainly written in Java. Before the template detection process, syntactical errors of the example pages are fixed up and the Firefox layout engine computes the visual layout properties of tags.

To achieve a fair comparison with other methods, we choose two third-party datasets as our test bed. The first collection is the datasets used within EXALG[5], which contains 45 web sites. For each site, we randomly choose 10 pages to set up an extraction task and manually label the data fields and records (if the example pages are list pages). The other collection is the *Testbed for Information Extraction from Deep Web* (TBDW v1.02[6]), which contains 51 sites (5 pages per site) and the creators have manually labeled all the data fields of the records. Because our approach perform a page-level extraction, we also manually label those data fields out of the records, such as page title, search matches, time usage, etc. EXALG datasets contain both two types of pages, while TBDW datasets contain list pages only.

7.2 Evaluation metrics

We use the standard metrics *recall* and *precision* to evaluate the performance of our system. For data field identification, recall and precision are defined below:

$$field\text{-}recall = \frac{E_{f\text{-}correct}}{N_{f\text{-}total}} \quad \text{and} \quad field\text{-}precision = \frac{E_{f\text{-}correct}}{E_{f\text{-}total}},$$

where $E_{f\text{-}correct}$ is the total number of correctly identified data fields, $E_{f\text{-}total}$ is the total number of identified data fields and $E_{f\text{-}total}$ is the total number of manually labeled data fields. Since some related works (e.g., ViNTs[7]) only extract records from list pages, we evaluate our system in record-level extraction as well. The meanings of symbols that occur in record level metrics below are similar to the corresponding symbols defined above.

$$record\text{-}recall = \frac{E_{r\text{-}correct}}{N_{r\text{-}total}} \quad \text{and} \quad record\text{-}precision = \frac{E_{r\text{-}correct}}{E_{r\text{-}total}}.$$

7.3 Experimental results and discussion

We compare our approach with EXALG[4], ViNTs[7] and ViPER[8] on the two datasets. The results are shown in Table 3. The running results of the other three systems are directly acquired from the papers above. For the EXALG datasets, we compare our approach with EXALG in field-level extraction. Table 3 shows that Grubber improves the performance remarkably. We conclude two reasons to the improvement: ① more characteristics of underlying structure of template-generated web pages are explored; ② more heuristics are used in Grubber, for example, the visual layout information (e.g., the width and height of HTML tags in visual), the attributes of HTML tags, string characteristics and etc. For the TBDW datasets, we compare our approach with other two methods in record-level extraction. Table 3 shows we gain a better performance than ViNTs. Comparing with ViPER, our system achieves almost the same perfect performance. From the table, we also can see that Grubber works better on TBDW datasets than EXALG datasets. This is because EXALG datasets have web sites with detail pages, while TBDW datasets consist of only list pages.

Table 3 Experimental results on two third-party test beds

	Datasets used within EXALG		TBDW v1.02			
Web sites	45 (450 pages)		51 (255 pages)			
Methods	EXALG	Grubber		ViNTs	ViPER	Grubber
Field-Recall (%)	80	91.5	Record-Recall (%)	89.2	97.6	98.2
Field-Precision (%)	–	95.3	Record-Precision (%)	93.5	98.5	98.6

Table 4 shows the experimental results on the two types of pages. As we can see in this table, our system can achieve high extraction accuracy in both two types of pages. Compared to detail pages, our system work better for list pages. This is due to two reasons: ① list pages usually have simple web data schemas; ② heuristics such as structural and visual layout characteristics are more effective in list pages.

Table 4 Experimental results on list and detail pages in Grubber

	List pages	Detail pages
Field-Recall (record-recall) (%)	97.1 (98.6)	90.2
Field-Precision (record-recall) (%)	97.8 (98.7)	92.8

8 Related Works

Many approaches have been reported in the literature for information extraction from web pages. These can be classified along different dimensions: the targeted information source (free text vs. generated by template; list pages vs. detail pages), degree of automation (manual, semi-automatic vs. fully automatic), complexity of data extracted (flat vs. nested), extraction level (field vs. record). Detailed discussions of various approaches can be found in several surveys[1,3]. We now discuss some of the most closely related works.

In the sense of task domain, degree of automation and extraction level, the most relevant work to our approach are RoadRunner[9], EXALG[4], which are fully automatic and applicable to both list pages and detail pages. RoadRunner starts off with the first input page as its initial template, then for each subsequent page, it uses a matching algorithm to compare the page with the current template and adjusts the current template when mismatches happen. The full algorithm has an exponential time complexity. Moreover, RoadRunner assumes the templates are *union-free* regular expressions, which does not hold for many collections of web pages. EXALG is an improved approach and can overcome the main limitations of RoadRunner. It imports two novel techniques, differentiating roles and equivalence classes to help detect the unknown template. In the first technique, the occurrence paths are used to differentiate roles of the tokens. In the second technique, the occurrence frequencies of the tokens over the input pages (occurrence vector) are used to find equivalence classes. Inspired by EXAG, our

approach also utilizes the occurrence paths and the occurrence frequencies of the tokens. However, our approach has several major differences with EXALG: Firstly, EXALG treats the input pages as token sequences and considers little about the characteristic of tree structure of HTML document (just using the paths of tokens). Secondly, EXALG tries to group the tokens by differentiating roles, while our approach groups the tokens using a set of defined rules and an average linkage clustering method based on the similarity of token trees. Thirdly, several effective features are ignored in EXALG, e.g. visual layout information, tag attributes and string similarity.

In the sense of the techniques used, we focus on how to discover the repeated patterns from the example pages in unsupervised systems. Besides the approaches discussed above, other relevant works are DeLa[10], ViNTs[7], ViPER[8], ViDRE[11], DEPTA[12], etc. DeLa[10] treats the input pages as strings and employs an algorithm to discover the continuously repeated (C-repeated) substrings using suffix trees. Visual features and tree structure of HTML document are ignored in DeLa. ViNTs[7] proposes an algorithm to find *SRRs* (search result records) from returned pages of the search engines. It treats a web page as a collection of *content lines* and identifies those content lines that separate the web page into several *blocks*. Then blocks are grouped by their visual similarities for subsequence analysis. ViNTs works best on search engine results. However, it requires a no-result page and some special heuristics of SRRs are used. ViPER[8] identifies and ranks potential repeated patterns using visual features. Then matching subsequences of the pattern with the highest weight is aligned with global multiple sequence alignment techniques. ViDRE[11] first builds visual block tree from the input page and discovers those visual blocks each of which contains a data record. Many features like position, layout, font attributes are used as heuristics to perform the task. DEPTA[12] applies a partial tree alignment technique to mine data records in a web page. Both tree edit distance and visual features are used to discover data records. However, the approaches above are applicable to list pages, while our approach can be used for both two types of pages directly.

9 Conclusion and Future Work

This paper proposed a novel approach to automatically detecting templates from a set of example pages and extracting data in field level. We formalize the template detection problem and make an analysis of the underlying structure of template-generated pages. Our approach detects the unknown template by two algorithms. The first algorithm is to group the tokens in the pages into CTokens, using a set of defined rules and an average linkage clustering method based on the similarity of token trees. The second algorithm is to construct the target template by analyzing the occurrence patterns of the CTokens. The statistical, structural and visual layout features of template-generated pages are used as the heuristics in the algorithms. We have built a prototype of a wrapper generation system called Grubber and experimental results show that our approach can achieve high extraction accuracy in both list and detail pages.

There are several important issues remaining to be addressed in our subsequent research: ① improving the performance of detecting templates using more features, e.g. font attributes (such as size, color, etc.); ② enhancing the usability by supporting assistant utilities, e.g. republishing the wrappers generated from detected templates as web services.

References:

[1] Chang CH, Kayed M, Girgis MR, Shaalan K. A survey of Web information extraction systems. IEEE Trans. on Knowledge and Data Engineering, 2006,18(10):1411−1428.

[2] Gold ME. Language identification in the limit. Information and Control, 1967,10(5):447−474.

[3] Laender AHF, Ribeiro-Neto BA, da Silva AD, Teixeira JS. A brief survey of Web data extraction tools. SIGMOD Record, 2002,31(2):84−93.

[4] Arasu A, Hector GM. Extracting structured data from Web pages. In: Proc. of the ACM SIGMOD Int'l Conf. on Management of Data. San Diego: ACM Press, 2003. 337−348.

[5] EXALG datasets. http://infolab.stanford.edu/~arvind/extract/

[6] TBDW v1.02. http://daisen.cc.kyushu-u.ac.jp/TBDW/testbed/

[7] Zhao HK, Meng WY, Wu ZH, Raghavan V, Yu C. Fully automatic wrapper generation for search engines. In: Proc. of the 14th Int'l Conf. on World Wide Web (WWW 2005). Chiba: ACM Press, 2005. 66−75.

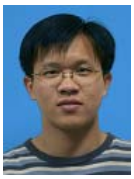
[8] Simon K, Lausen G. ViPER: Augmenting automatic information extraction with visual perceptions. In: Proc. of the ACM CIKM Int'l Conf. on Information and Knowledge Management. Bremen: ACM Press, 2005. 381−388.

[9] Crescenzi V, Mecca G, Meraldo P. RoadRunner: Towards automatic data extraction from large Web sites. In: Proc. of the 27th Int'l Conf. on Very Large Data Bases (VLDB 2001). Roma: Morgan Kaufmann Publishers, 2001. 109−118.

[10] Wang JY, Lochovsky FH. Data extraction and label assignment for Web databases. In: Proc. of the 12th Int'l World Wide Web Conf. (WWW 2003). Budapest: ACM Press, 2003. 187−196.

[11] Liu W, Meng XF, Meng WY. Vision-Based Web data records extraction. In: Proc. of the 9th SIGMOD Int'l Workshop on Web and Databases (WebDB 2006). Chicago: ACM Press, 2006.

[12] Zhai YH, Liu B. Structured data extraction from the Web based on partial tree alignment. IEEE Trans. on Knowledge and Data Engineering, 2006,18(12):1614−1628.



YANG Shao-Hua was born in 1981. He is a Ph.D. candidate of the Institute of Computing Technology, the Chinese Academy of Sciences. His current research areas are Web mining and service-oriented computing.

LIN Hai-Lue was born in 1982. He is a Ph.D. candidate of the Institute of Computing Technology, the Chinese Academy of Sciences. His current research areas are Web information retrieval and service-oriented computing.

HAN Yan-Bo was born in 1962. He is a professor and doctoral supervisor at the Institute of Computing Technology, the Chinese Academy of Sciences, and a CCF senior member. His research areas are software integration and service grid.