九州大学学術情報リポジトリ
Kyushu University Institutional Repository

A Template Discovery Algorithm by Substring Amplification

Ikeda, Daisuke
Computing and Communications Center, Kyushu University

Yamada, Yasuhiro
Department of Informatics, Kyushu University

Hirokawa, Sachio
Computing and Communications Center, Kyushu University

https://hdl.handle.net/2324/2815

出版情報：DOI Technical Report. 220, 2003-12. Department of Informatics, Kyushu University
バージョン：
権利関係：

KYUSHU UNIVERSITY

A Template Discovery Algorithm by Substring Amplification

Daisuke Ikeda[1], Yasuhiro Yamada[2], and Sachio Hirokawa[1]

[1] Computing and Communications Center, Kyushu University
{daisuke, hirokawa}@cc.kyushu-u.ac.jp
[2] Department of Informatics, Kyushu University
y-yamada@i.kyushu-u.ac.jp
yshiro@matu.cc.kyushu-u.ac.jp

Abstract. In this paper, we consider to find a set of substrings common to given strings. We define this problem as the *template discovery problem* which is, given a set of strings generated by some fixed but unknown pattern, to find the constant parts of the pattern. A *pattern* is a string over constant and variable symbols. It generates strings by replacing variables into constant strings. We assume that the frequency distribution of replaced strings follow a power-law distribution. Although the longest common subsequence problem, which is one of the famous common part discovery problems, is well-known to be NP-complete, we show that the template discovery problem can be solved in linear time with high probability. This complexity is achieved due to the following our contributions: reformulation of the problem, using a set of substrings to express a string, and counting all occurrences $F(f)$ with frequency f instead of just frequency f. We demonstrate the effectiveness of the proposed algorithm using data on the Web. Moreover, we show noise robustness and effectiveness even when input strings are generated by a union of patterns and pattern with the iterate operation.

1 Introduction

The progress of computer technologies makes it easy to make and store large amount of text files such as HTML/XML files, Emails, Net News articles, genome sequences, etc. It is an important problem to find some rule common to given text files.

A subsequence is often used to express a common part. A *subsequence* of some string w is a string obtained by deleting zero or more symbols in w. For example, *cbbbab* is a subsequence of *ccbbcbabbabca*.

To find the longest common subsequence is known to be NP-complete [11]. An input for this problem is a set of arbitrary strings. However, when we want to examine a set of strings, we generally expect and assume that there exist common parts in the strings. For example, when DNA sequences are given, the sequences are collected carefully in advance, so that many of them have common parts. In the field of Web mining, information extraction has been well studied [3, 9, 10, 19]. It is a problem to find a common part and then extract contents embedded among common parts. In this case, an input for an extraction algorithm is a set of HTML files generated from some fixed template. Therefore, we can assume that input strings for a common part discovery algorithms are positive example.

Next, we revisit a subsequence which is used to express a common part. In the above example, we mentioned that *cbbbab* is a subsequence of *ccbbcbabbabca*. However, we can also underline *ccbbcbabbabca*. A subsequence of a string is concatenated by ordered substrings of the string, and generally there exist different substrings which comprise the same subsequence. This is not suitable when functional segments of genome sequences must be extracted. In case of information extraction, it is not natural that a template are different among given HTML files.

Next, we consider uncommon parts. In the genome sequence, it is assumed that uncommon substrings are not so important for functionalities of genes and they are changed (nearly) randomly during a long time of evolution. In the case of information extraction, words or sentences of natural languages are embedded as contents in uncommon

parts. It is empirically known that the frequencies of words follow a power-law distribution in the case of Western languages. This phenomena is known as Zipf's law. Thus, we assume that the frequencies of uncommon substrings follow a power-law distribution (see Section 5.1).

From above observations, we define common parts discovery as the *template discovery problem*. Given some strings generated by some fixed but unknown pattern, the problem is to find the constant strings of the pattern. A *pattern* is a string over constant and variable symbols. It generates words by replacing variables in it into constant strings. Frequencies of replaced strings are assumed to follow a power-law distribution.

A pattern language has been well studied in the context of machine learning. It is known that there is no efficient learning algorithm unless we restrict the number of variable's occurrences or the number of different variables [2, 8, 17]. In the viewpoint of information extraction, however, these restrictions are impractical because one type of a variable corresponds to one type of contents. These restrictions are crucial especially when we treat a semi-structured data which has a tree structure and a node of the tree may have many children of the same type. On the other hand, in the setting of the template discovery problem, only constant strings are discoverd, but there is no restriction on the number or type of variables.

From the definition of the problem, we expect that we can solve the problem by using the disparity of substring frequencies between the template and the other parts. However, the most frequent substring is a string with length 1, that is, a character. To avoid this problem in N-gram statistics of the natural language processing, the length N is determined carefully in advance [14, 16]. In the setting of frequent pattern mining, which is a major framework of data/text mining, the minimum support must be given to a mining algorithm [1, 4, 5].

In this paper, we propose an algorithm for the template discovery problem. It does not require any additional inputs such as negative (or background) examples, background knowledge, or parameters. The algorithm requires positive example only. Instead, we assume that constant strings must be enough long. The algorithm exploits the assumption that frequencies of replaced strings follow a power-law distribution and solves the template discovery problem with high probability.

The time complexity of the algorithm is $O(n)$, where n is the total length of input strings. To develop such a speedy algorithm, we utilize (1) a method[3] to express a constant string by a set of substrings of it, (2) a method, called the *substring amplification*, in which we count the total number $F(f)$ of occurrences of strings appearing exactly f times instead of the frequency (the number of occurrences) f. This algorithm makes full use of the fact that the frequencies of substituted strings follow a power-law distribution and it discovers the template with high probability.

We implemented the algorithm and gave HTML files collected from the Web into the algorithm. Experimental results exhibits that our algorithm well discovers the templates even if many noise files are included, a set of input files has multiple templates, or input files are generated by a union of different patterns or a pattern with the iterate operation.

2 Preliminaries

2.1 Pattern Language

Σ is a finite alphabet. Σ^* denotes the free monoid over Σ. An element of Σ^* is called a *string*. For a string x, if there exist strings $u, v, w \in \Sigma^*$ such that $x = uvw$, we say that v (resp. u and w) is a *substring* (resp. a *prefix* and a *suffix*) of x.

$V = \{x_1, x_2, \ldots, \}$ is a infinite set of symbols disjoint from Σ. An element in V is called a *variable*. We sometimes call a string over Σ a *constant* string in contrast with a variable. A *pattern* is a string over $\Sigma \cup V$. A pattern is *regular* if it has at most one occurrence for each variable.

[3] Originally, a similar method was introduced in [7].

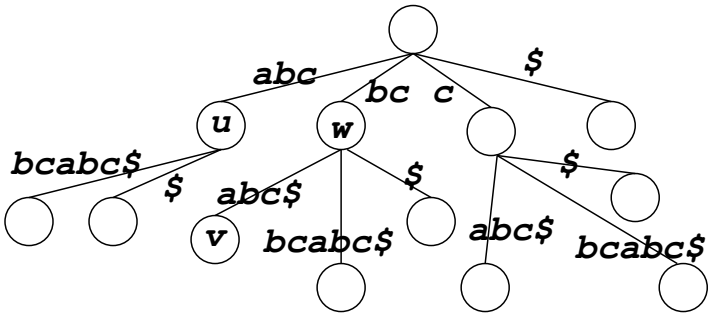


Fig. 1. An example of a suffix tree for *abcbcabc*

A *substitution* is a homomorphism with respect to concatenation from V to Σ^*. A substitution θ which maps x to u is denoted by $[x := u]$. We abbreviate substitutions which map x_i to u_i $(1 \leq i \leq m)$ to $[x_1 := u_1, \ldots, x_m := u_m]$.

For a pattern p and a substitution $\theta = [x_1 := u_1, \ldots, x_m := u_m]$, $p\theta$ is the string obtained by replacing all occurrences of x_i in p with u_i for each $1 \leq i \leq m$. The *language* of p, denoted by $L(p)$, is $\{w \in \Sigma^* \mid \exists\theta; p\theta = w\}$.

A finite set $S \subset \Sigma^*$ of strings is called a *sample*.

2.2 Suffix Tree

Let \$ the special character such that $a \neq \$$ for any $a \in \Sigma$ and $\$ \neq \$$. For a string w, $A = w\$$ and an integer p $(1 \leq p \leq |A|)$, A_p denotes A's suffix starting at the pth character of A. Let $A_{p_1}, A_{p_2}, \ldots, A_{p_n}$ be all suffices of A in lexicographic order[4]. The *suffix tree* for w is the compact trie for $A_{p_1}, A_{p_2}, \ldots, A_{p_n}$. For each node v of the tree, *string*(v) denotes the string obtained by concatenating all strings labeled on edges of the path from the root to v. We call *string*(v) a *branching string*.

We assume that, for each node v of the tree, its children are sorted in lexicographical order of strings labeled on edges between v to them.

Example 1. Fig. 1 is the suffix tree for $s = abcbcabc$. For nodes u and v in Fig. 1, we have *string*(u) = *abc* and *string*(v) = *bcabc*\$.

Let u be a node of a suffix tree. The number of occurrences of *string*(u) equals to the number of leaves below u. In the above example, node w has three leaves and *string*(w) appears three times in *abcbcabc*.

Let v be the parent node of u. Then, *string*(v) is a proper prefix of *string*(u) from the definition of the branching strings. Moreover, let w be a prefix of *string*(u) which includes *string*(v) as a prefix. Then, the numbers of occurrences of w and *string*(u) are the same. For example, both *string*(u) = *abc* and its prefixes *a*, *ab* appear two times. Therefore, when we count substring frequencies, all we have to do is to count only branching strings.

3 Template Discovery Problem

In this section, we discuss the common part discovery from the viewpoint of information extraction and then formulate the template discovery problem.

[4] We define that x is the predecessor of $x\$$ for any $x \in \Sigma^*$ in the order.

An input for information extraction is a set of files which share the same style and format, and uncommon parts are extracted as contents. In the pattern language, contents are substituted into variables and the common parts are the constant strings. Therefore, first we have to find common constant strings as the parts.

One variable may appear twice or more times in a pattern since the same content is sometimes embedded iteratively in one of input files. In the field of machine learning, many classes of pattern languages in which the occurrence of each variable is not limited have been studied. However, the objective of this paper is to extract all contents but not to check whether some of the contents are the same or not since the latter task is a kind of schema extraction which should be the next step of information extraction. In this sense, we restrict a pattern to be regular.

Because each variable appears at most once in a regular pattern p, a successive sequence of variables can be replaced by new one variable. Thus, p is denoted uniquely as follows: $p = w_0 x_1 \cdots w_{m-1} x_m w_m$ ($w_i \in \Sigma^+$ $(0 \leq i \leq m)$, $x_i \in V$ $(1 \leq j \leq m)$). Then, a set $\{w_0, \ldots, w_m\}$ of constant strings of p is called a *template*, where at least one constant string is not the empty string. From this definition, at least one constant string is included in a template.

Note that the order of w_i is not considered. This is because it is not known whether there is a polynomial time learning algorithm even if we restrict a pattern to be regular [18] but if we can find constant strings with their order, we can construct the corresponding regular pattern by inserting mutual variables between found constant strings. From the viewpoint of information extraction, all we have to do is to extract all contents. So, we do not need their order.

When we consider information extraction, contents are embedded between constant strings. Usually, the contents are natural strings such as words, phrases, or sentences written in natural languages. Thus, we assume that the substring frequencies in substituted strings obey a power-law distribution. Zipf's law is a power-law distribution.

Let $V(f)$ be the number of substrings which appear exactly f times in a given sample S. Then, Zipf's law says that $V(f) = bf^{-a}$ for some constant a and b, or

$$\log V(f) = b - a \log f. \tag{1}$$

Since the above equation holds *on average*, the frequencies of some substrings happens to deviate from the equation.

From the above observations, we formulate the template discovery problem as follows:

Definition 1 (Template discovery problem). *The template discovery problem is, given a sample S which is generated by some fixed but unknown pattern by substitutions following a power-law distribution, to find a template of the pattern.*

Note that the distribution itself is not given.

The learning problem of the pattern language is to find a pattern descriptive to a given sample [2]. A pattern p is *descriptive* to a sample S if $S \subseteq L(p)$ and there is no pattern q such that $S \subseteq L(q) \subset L(p)$. On the other hand, the above definition say that we have to find a pattern which generates a given sample.

In general, there exist several descriptive patterns for some sample S. For example, both $p_1 = xbcybcz$ and $p_2 = xbcay$ are descriptive to $S = \{abcabc, bcbca\}$, and we can not conclude that which pattern generates S. If p_1 generates S, then (bc, bc) is the sequence of all constant strings. In this case, a is substituted after bc for all elements in S, so bca becomes to be common to them. From the practical viewpoint, however, with large size of $|S|$ and $|\Sigma|$, there few possibilities for such substitutions.

When we consider information extraction, a constant string rarely happens to be substituted into a variable, since each constant string have some adequate length. Moreover, when we consider HTML/XML as targets of information extraction, "<" or ">" is not included in the contents, so such substitutions do not happen. From these observations, we expect to find a template of a pattern which generates S with high probability.

4 Template Discovery with Substring Amplification

In this section, we present an algorithm for the template discovery problem and show that the algorithm solves the problem in linear time with high probability.

We assumed that values of $V(f)$ follow a power-law distribution, where $V(f)$ is the number of different substrings appearing in S. Even if we fix the size of S and the lengths of strings in S, the different substrings in S is subject to occurrences of characters. On the other hand, the all occurrences of substrings is decided uniquely if the length of the target string is fixed. Therefore, we examine the relationship $V(f)$ and $F(f)$ which is all occurrences of substrings which appear exactly f times in S.

First, we estimate the number of different frequencies.

Lemma 1. *Let n be the total length of the strings in a sample S. Then, the number of different frequencies of substrings appearing in S is at most $O(n)$.*

Proof. In a suffix tree, a node corresponds to a frequency. There exist at most $O(n)$ different frequencies since the number of nodes in a suffix tree is $O(n)$.

Next, we estimate $G(f) = F(f)/F(f-1)$ and show that values of $G(f)$ converge to be a constant asymptotically if values of $V(f)$ follow a power-law distribution.

Lemma 2. *If $V(f)$ satisfies Equation (1), then $G(f) = F(f)/F(f-1) = (1-1/f)^{a-1}$.*

Proof. From Equation (1), we have

$$\begin{aligned} \log V(f) &= b - a \log f \\ \log f + \log V(f) &= b + (1-a) \log f \\ \log fV(f) &= b + (1-a) \log f. \end{aligned}$$

Since $F(f) = fV(f)$, we have

$$\log F(f) = b + (1-a) \log f.$$

If $a = 1$, $F(f)$ become to be a constant. Therefore, we can assume $a \neq 1$ without loss of generality.

Let $A = a - 1$, $B = b$. Then, $F(f) = Bf^{-A}$ and $G(f)$ is calculate as follows:

$$\begin{aligned} G(F) = \frac{F(f)}{F(f-1)} &= \frac{(f-1)^A}{B} \cdot \frac{B}{f^A} \\ &= \left(\frac{f-1}{f}\right)^A \\ &= (1-1/f)^A. \end{aligned}$$

Thus, $G(f)$ becomes asymptotically to be a constant as f is increasing if values of $V(f)$ follow a power-law distribution.

Let $t = \{w_0, \ldots, w_m\}$ be a template. Then, there exist at least $O(l^2)$ occurrences of substrings for each constant string, where l is the minimum length of constant strings. For simplicity, we now assume that all characters in a constant string are different. In this case, both a constant string and any substring of it appear exactly $|S|$ times. Therefore, $F(|S|)$ is at least $O(m|S|l^2)$. On the other hand, values of $F(|S|+1)$ and $F(|S|-1)$ follow a power-law

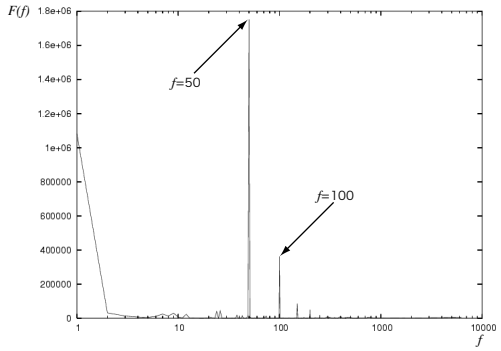


Fig. 2. Peaks in $F(f)$ graph

```
function Template (var S: sample): set of strings
  var
    V, F: hash table;
  begin
    V:=Count(S); {count all branching strings}
    for f in keys(V); {for all frequencies}
       F(f):=0;
       for w in V(f);
         F(f) += f|w|;
       end;
    end ;
    f:=FindPeaks(F);
    return(V(f));
  end ;
```

Fig. 3. Template discovery algorithm which implements the substring amplification

distribution. Thus, the value $F(|S|)$ is quite large compared to those of $F(|S|+1)$ and $F(|S|-1)$ if $|S|$, m, and l are enough large.

For example, let S be a set of 50 HTML files collected from an online news outlet. We have a graph of Fig. 2 by plotting $F(f)$ for each f in log scale. We see peaks at $f = c|S|$ for $c = 2, 3, \ldots$ as well as $f = |S|$ since there are substrings which appear twice or more in constant strings in these files.

From these observations, we define a peak as follows:

Definition 2. *Let $G(f) = F(f)/F(f-1)$. Then, we say that a frequency f_p provides a* maximal peak *if $G(f_p)$ is maximum among all values of $G(f)$. We also say that $f = c f_p$ ($c = 1, 2, \ldots$) provide* peaks.

We can expect that constant strings are reconstructed from substring constituting peaks. A peak is constituted by adding occurrences of all substrings. So, we call this template discovery method the *substring amplification*.

The main algorithm for the template discovery problem is presented in Fig. 3. An input for the algorithm is a sample S and it outputs a set of strings. We will show that this output equals to a template which we want to find with high probability.

First, the algorithm calls `Count(S)`. This routine constructs a suffix tree T for $s_1\$s_2\$\cdots\$s_{|S|}$, where $S = \{s_1, s_2, \ldots, s_{|S|}\}$. Then, it counts frequencies of $string(v)$ for each node v in T. Then, it creates a hash table `V` in which a key is a frequency `f` and a value `V(f)` is the number of different branching words appearing exactly `f` times.

Next, the algorithm calculates `F(f)` from `V(f)`. Although $F(f) = f \cdot V(f)$, we need to count non-branching strings since `V` contains frequencies for only branching strings. Therefore, we first initialize `F(f):=0`, and then `F(f) += f|w|` for each branching word w and its frequency f.

Next, the algorithm calls `FindPeaks(`F`)` which returns a frequency providing a maximal peak. This is done by calculation of $F(f)/F(f-1)$ for each $f \geq 2$.

Finally, the algorithm returns a set `V(f)` of branching strings such that they appear exactly f times and f provides the maximal peak.

We show the correctness of algorithm `Template`. The algorithm sometimes fails to find a constant string or outputs non-constant strings wrongly depending on substituted strings. Therefore, we estimate the error probability when a probability distribution for all characters is given.

Although a uniform distribution over Σ is one of basic distributions, it is known that frequencies of substrings over this distribution do not follow a power-law distribution. On the other hand, for characters in English sentences, frequencies of characters follow a power-law distribution as well as frequencies of English words [15]. Moreover, if requencies of characters in Σ follow a power-law distribution, it is also known that frequencies of strings over Σ also follow the distribution. Therefore, in the following theorem, we assume that all characters in substituted strings appear independently and frequencies of them follow a power-law distribution.

Theorem 1. *Let t_0 be a template whose corresponding regular pattern p generates a given sample S. We assume that any string in t_0 is not included in any other string in t_0. Then, Algorithm `Template` in Fig. 3 outputs t_0 with high probability. The error probability is bounded by $1/c^{rk} + 1/c^{l_0} + 3/c^{|S|}$, where S is a sample, l_0 is the minimal length among the constant strings in t_0, $0 < 1/c < 1$ is the highest probability assigned to a character in Σ, and r and k satisfies $m_0|S|l_0^2/F(|S|-1) < rk^2/F(r-1)$.*

Proof. Let $t_0 = \{w_0^0, \ldots, w_{m_0}^0\}$ and $t = \{w_0, \ldots, w_m\}$ be a set of strings output by `Template`.

(1) First, we consider that `Template` outputs a string which is not a constant string. We have to consider the following two cases in which t includes a string not including in t_0: (1-1) `Template` output $f \neq |S|$ as the maximal peak while the maximal peak must be $|S|$ and (1-2) there exist a string which is not included in t_0 but appears exactly $|S|$ times. The latter case happens with probability $1/c^{|S|}$ since this holds if the length of the string is 1.

The former case occurs when a long string appears frequently in substituted strings. In this case, the error probability is one that a string w^k with length k appears r times in the substituted strings. This probability is $P^r(w^k) = 1/c^{rk}$, where $G(|S|) = F(|S|)/F(|S|-1) < F(r)/F(r-1) = G(r)$ must hold since r must provide the maximal peak. Since total occurrences of substrings in w^k is least $O(rk^2)$, $F(r) = O(rk^2)$. Therefore, the following equation must be satisfied.

$$G(|S|) = F(|S|)/F(|S|-1) < F(r)/F(r-1) = G(r),$$
$$m_0|S|l_0^2/F(|S|-1) < rk^2/F(r-1).$$

(2) Next, we check that `Template` outputs all constant strings in t_0. In this case, we also have to consider the following two cases: (2-1) a string which equals to w_i^0 for some i is included in a substituted string and (2-2) a character is substituted just before (or after) w_i^0 for some i for all strings in S.

(2-1) For $j \neq i$, w_j^0 appears exactly $|S|$ times and $|S|$ provides the maximal peak. However, t does not contain w_i^0 since w_i^0 appears $|S|+1$ times. This case happens with probability $P(w_i^0) = 1/c^{l_0}$.

(2-2) Assume that a string w^k with length k is substituted before w_i^0. In this case, $w_i = w^k w_i^0$ and the frequencies of w^i and w_i^0 are the same. Therefore, w^i is detected as a constant string by `Template` instead of w_i^0. This holds when

$k = 1$. The probability for this is $2/c^{|S|}$ since this is the same as the probability in which the same character appears just before or after $|S|$ times.

Thus, algorithm `Template` solve the template discovery problem with the error probability at most $1/c^{rk} + 1/c^{l_0} + 3/c^{|S|}$, where r and k satisfies $m_0|S|l_0^2/F(|S|-1) < rk^2/F(r-1)$. So, if $|S|$, l_0, and $|\Sigma|$ are enough large, the error probability is close to 0 asymptotically.

In the above proof, we only used the property that each character appears independently. Therefore, the above theory holds for any distribution such as uniform and power-law distributions, if character independence is guaranteed.

We also assumed that $w_i \in t_0$ is not included in $w_j \in t_0$ $(i \neq j)$. Algorithmically, we can easily remove this assumption, but this increases the difficulty to define a peak. In Definition 2, we defined the maximal peak and `Template` only finds it. If $w_i = w_j$ for some $i \neq j$, `Template` have to find another peaks at $f = 2|S|$. Similarly, we have to consider $f = c|S|$ $(c = 3, 4, \ldots)$. In this case, we have to introduce some threshold value. This also holds for more complex patterns which will be described in the next section.

Next, we show the time complexity of the algorithm.

Theorem 2. *The time complexity of* `Template` *is* $O(n)$*, where* n *is the total length of the strings in a given sample* S*.*

Proof. `Count()` constructs a suffix tree in $O(n)$ time [12]. It counts frequencies by traversing nodes on T. This counting is done in $O(n)$ time since there is $O(n)$ nodes in a suffix tree.

It takes also $O(n)$ time to compute $F(f)$ since calculation $F(f) += f \cdot |w|$ is totally executed as many times as the number of branching strings.

`FindPeaks()` returns a frequency f providing the maximal peak by computing $G(f) = F(f)/F(f-1)$ for each frequency. From Lemma 1, there exist at most $O(n)$ different frequencies. Therefore, this routines done in $O(n)$ time. This completes the proof.

5 Experiments

First, using novels collected from the Web, we examine the validity of the assumption that the frequencies of substituted substrings follow a power-law distribution.

In Definition 1, an input for the problem is a set of strings generated by a single regular pattern. However, noise files are included or some files are generated by some more complex patterns in practice. Therefore, next, using HTML files on the Web, we show that the substring amplification finds peaks constituted by substrings in template substrings even when a sample is generated by a unison of multiple patterns or a sample includes many noise data as well as being generated by a single pattern.

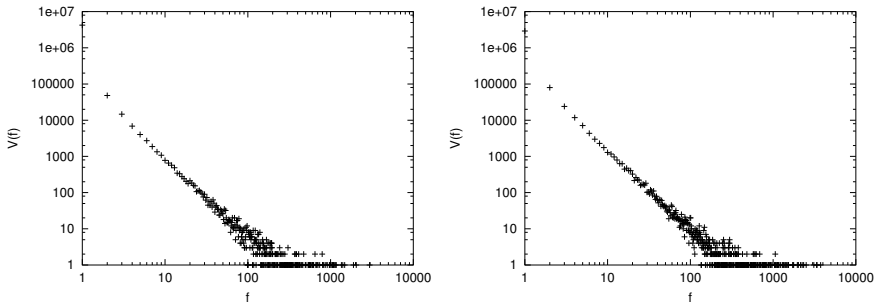
More precisely, we introduce additional operation “union” and “iteration” and redefine an input sample for the template discovery problem as the set of strings generated by a new pattern.

Now, a sample may be generated by multiple templates. Therefore, we have to consider non-maximal peaks. Although the maximal peak is uniquely decided, it is difficult problem to decide a peak theoretically since the hight of peaks heavily depends on the number of input strings, a length of constant strings of patterns, and so on. Experiments in this section, the authors decide which frequencies provide peaks.

5.1 Power-Law Distribution

We assumed that values of $V(f)$ follow a power-law distribution in the definition of the template discovery problem. In this section, we examine the frequency f and $V(f)$ using documents without explicit templates [20].

Fig. 4 is two graphs of $V(f)$ which is the number of different substrings appearing exactly f times. We use (a) a



(a) Japanese Novel (b) English Novel

Fig. 4. Graphs of $V(f)$, the number of different substrings which appear exactly f times

Japanese novel "Kokoro[5][6]" written by Souseki Natsume (487 KBytes) and (b) an English novel "Metamorphosis[7]" written by Franz Kafka (134 KBytes).

From Fig. 4, we see that $\log V(f)$ is proportional to $\log f$ for both languages and satisfies Equation (1).

Although Zipf's law is originally known for grammatical words or strings with fixed length, $V(f)$ is calculated from all occurrences of substrings whose lengths are less than 50. Fig. 4 exhibits that Zipf's law holds even for such a case.

5.2 Single Template

In this section, we show an experiment using files which seem to be generated by a single template. These files are collected from "Sankei Shimbun[8]". They are 50 files (526 Kbytes). Fig. 2 is the $F(f)$ graph for these files.

In this figure, we see sharp peaks at $f = 50, 100, 150, 200$ and so on. The maximal peak is at $f = 50$. These peaks tell that there exists a common template in each file and consequently substrings in the template appear exactly 50 times. In fact, substrings appearing exactly 50 times well cover the template.

Some substrings in the template appear two or more times. These substrings constitute other peaks. For example, "<title>" or "</title>" appear at most once in a file, but, "title" appears twice. Thus, "title" appears exactly 100 times in all files. Such more frequent substrings constitute smaller peaks because they are shorter than substrings appearing exactly 50 times.

5.3 Multiple Templates with Different Occurrence Ranges

Next, we show an experiment for search result pages of Yahoo![9]. Search result pages, in general, contain at least two different types of templates. One is a template for each file and the other is a template for each search result which usually appears 10 or 20 times in a file.

[5] http://www.aozora.gr.jp/
[6] It stands for "heart" or "spirit" in Japanese.
[7] http://www.promo.net/pg/
[8] http://www.sankei.co.jp/
[9] http://www.yahoo.com/

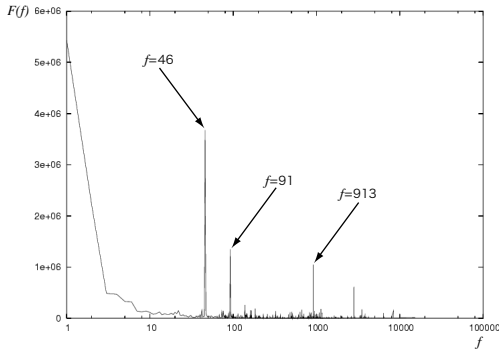


Fig. 5. f vs. $F(f)$ graph for search result pages of Yahoo!

We used a category name of Yahoo! for a query to get search results. We chose randomly 50 names among all category names, then we gave each name as a query. We failed to get search results for 4 category names, consequently we collected 46 files. Basically each file contains 20 search result pages, but one file contains only 14 result pages and another one only 19 result pages. Thus, totally 46 files (1212 Kbytes) contain 913 search result pages.

Fig. 5 is the $F(f)$ graph for these files. In Fig. 5, we see the maximal peak at $f = 46$ which is the same as the number of files. We see another peak at $f = 91$ which is close to $f = 92 = 46 \times 2$. The authors expected that a substring of a constant string appears exactly two time in most of files but it appears only once in some file. Contrary to our expectation, this peak at $f = 91$ exhibits an evidence of existence for another template. A search result page of Yahoo! contains matched category names as well as matched Web pages. In this experiment, the number of matched category names was 91, so the template for these category names constitute the sharp peak.

We see another peak at $f = 913$ which is the number of search results. Therefore, in this experiment, we found three different templates whose periods are different. The number of templates are not fixed. This indicates that the substring amplification can treat a sample generated by a pattern with extra operation of iteration $(\cdot)^+$ like regular expressions.

5.4 Union of Multiple Patterns

Next, we show an experiment whose input sample is mix of three different templates. In addition to 50 files of Sankei Shimbun (Section 5.2), we use 104 files (3412 Kbytes) from asahi.com[10] and 140 files (2324 Kbytes) from Yomiuri Online[11]. Compared to the number of Sankei Shimbun files, the number of asahi.com files is about twice and one of Yomiuri Online files is about thrice.

Fig. 6 is the $F(f)$ graph for this sample. There exist maximal peaks for each news site. Thus, the substring amplification finds all templates if input files are generated from a union of patterns. The number of different patterns are arbitrary. This indicates that the substring amplification can treat a sample generated by a pattern with extra operation union $\cdot|\cdot$ like regular expressions.

We see other peaks in Fig. 6 at $f = 49$ and $f = 103$. These peaks are constituted by other templates which the authors did not expect. For example, asahi.com and Yomiuri Online independently have substrings which appear exactly 49 times.

[10] `http://www.asahi.com/`
[11] `http://www.yomiuri.co.jp/`

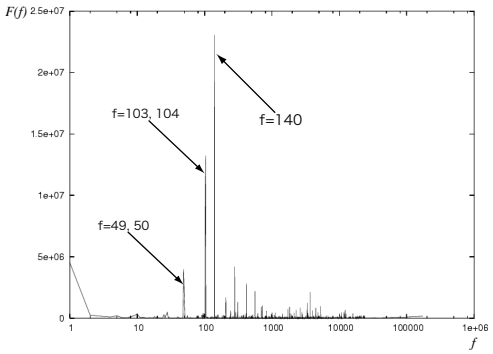


Fig. 6. f vs. $F(f)$ graph for Web pages mixed from 3 online news sites

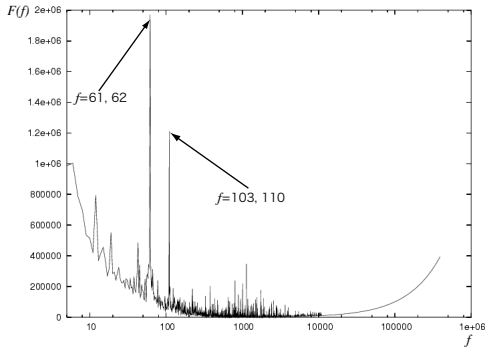


Fig. 7. f vs. $F(f)$ graph for Web pages in Kyushu University., where $f \geq 5$

5.5 Noisy Data

Next, we give 598 HTML files (5584 Kbytes) collected by following links at most three depth from the top page of Kyushu University[12].

Fig. 7 is the $F(f)$ graph for these HTML files. We see some high peaks. They are due to some HTML files which are modified version of the top page of the university. We see a peak clearly even if most of files are noise.

In general, files sharing with the template in a site are stored in one directory. So, we can collect such files from just the string processing of URLs. But, such a heuristics method does not guarantee that collected files are *all* files which share the template. On the other hand, the substring amplification collect *all* files sharing the template since it reconstruct constant strings from substring whose frequencies are the same.

6 Conclusion

We discussed common parts discovery in the viewpoint of information extraction and formulated it as the template discovery problem. The problem is, given a set of strings generated by some fixed but unknown pattern, to find a

[12] `http://www.kyushu-u.ac.jp/`

set of constant strings of the pattern. We assumed that the frequencies of substituted strings follow a power-law distribution and experimentally showed the validity of this assumption using Japanese and English novels.

We developed a method called *substring amplification* which is the key technology for the proposed template discovery algorithm. It amplifies the disparity of frequencies between constant strings and substrings in non-template parts. If we count just frequencies, we can not see high peaks among graph of f instead of $F(f)$ because frequencies of constant strings is subject to noise data and is buried in frequencies of noise and non-constant strings. Another problem to count just frequencies is how to decide the length of substring to be counted. In the framework of the substring amplification, we count *all* substrings and sum up them. Therefore, we see high peaks and we do no need to decide the length of substring to be counted. This means that a user of the proposed algorithm does not need to know about input data in advance.

We proved that the template discovery problem is solved in linear time by the proposed algorithm with high probability. However, when a substring of a constant string happens to be substituted, the algorithm finds it as a part of constant strings wrongly. Therefore, it is an important future work to define non-template parts as the contents which should be extracted and estimate recall and precision like information retrieval.

Experiments using data on the Web exhibited the effectiveness of the algorithm in the practical viewpoint. The algorithm found a template even if many of input files are noisy.

We also showed by experiments that the proposed algorithm found multiple templates if input strings are generate by patterns with the iteration and union operations like regular expressions. Since the class of languages generated by regular patterns is proper subclass of the class of regular languages, the expressive power of the regular patterns is less than one of the regular expressions. The regular expression is important as a theoretical background of XML [6, 13]. Thus, it is important that the substring amplification can treat such input strings.

However, this extension causes another problem. It is necessary to find multiples peaks when an input sample is a set of strings generate by patterns with extra operations described above. Therefore, we have to define some threshold for $G(f)$ to define *peak* exactly. This is an important future work.

We defined a template as a set of constant strings. It is challenging task to allow some mismatches in a constant string, so that some ambiguity is allowed, such as `[AC]` commonly used in genome informatics.

References

1. R. Agrawal, T. Imielinski, and A. Swam. Mining Association Rules between Sets of Items in Large Databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 207–216, 1993.
2. D. Angluin. Finding Patterns Common to a Set of Strings. *Journal of Computer and System Sciences*, 21:46–62, 1980.
3. A. Arasu and H. Garcia-Molina. Extracting Structured Data from Web Pages. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 337–348, 2003.
4. T. Asai, H. Arimura, T. Uno, and S. Nakano. Discovering Frequent Substructures in Large Unordered Trees. In *Proceedings of the 6th International Conference on Discovery Science*, Lecture Notes in Artificial Intelligence 2843, pages 47–61. Springer-Verlag, 2003.
5. J. Han, J. Pei, and Y. Yin. Mining Frequent Patterns without Candidate Generation. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 1–12, 2000.
6. H. Hosoya and B. C. Pierce. Regular Expression Pattern Matching for XML. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 67–80, 2001.
7. D. Ikeda, Y. Yamada, and S. Hirokawa. Eliminating Useless Parts in Semi-structured Documents using Alternation Counts. In *Proceedings of the 4th International Conference on Discovery Science*, Lecture Notes in Artificial Intelligence 2226, pages 113–127. Springer-Verlag, November 2001.
8. M. Kearns and L. Pitt. A Polynomial-Time Algorithm for Learning *k*-variable Pattern Languages from Examples. In *Proceedings of the 2nd Annual Workshop on Computational Learning Theory*, pages 57–71, 1989.
9. N. Kushmerick. Wrapper Induction: Efficiency and Expressiveness. *Artificial Intelligence*, 118:15–68, 2000.

10. N. Kushmerick, D. S. Weld, and R. B. Doorenbos. Wrapper Induction for Information Extraction. In *Proceedings of the 15th Intenational Joint Conference on Artificial Intelligence*, pages 729–737, 1997.
11. D. Maier. The Complexity of Some Problems on Subsequences and Supersequences. *J. ACM*, 25:322–336, 1978.
12. E. M. McCreight. A Space-Economical Suffix Tree Construction Algorithm. *J. ACM*, 23(2):262–272, 1976.
13. M. Murata. Extended Path Expressions for XML. In *Proceedings of the 20th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 126–137, 2001.
14. M. Nagao and S. Mori. A New Method of *N*-gram Statistics for Large Number of *n* and Automatic Extraction of Words and Phrases from Large Text Data of Japanese. In *Proceedings of the 15th International Conference on Computational Linguistics*, pages 611–615, 1994.
15. M. Nagao, S. Sato, S. Kurohashi, and T. Tsunoda. *Natural Language Processing*. IWANAMI KOZA Software Science 15. Iwanami Shoten, April 1996. (in Japanese).
16. T. Nakamura. Acquisition of Move Sequence Patterns from Encoded Strings of Go Moves. *Journal of Information Processing Society of Japan*, 43(10):3030–3036, October 2002. (in Japanese).
17. R. E. Schapire. Pattern Languages Are Not Learnable. In *Proceedings of the 3rd Annual Workshop on Computational Learning Theory*, pages 122–129, 1990.
18. T. Shinohara. Polynomial Time Inference of Extended Regular Pattern Languages. In *RIMS Symposium on Software Science and Engineering (1982)*, Lecture Notes in Computer Science 147, pages 115–127. Springer-Verlag, 1983.
19. Y. Yamada, D. Ikeda, and S. Hirokawa. Automatic Wrapper Generation for Multilingual Web Resources. In *Proceedings of the 5th International Conference on Discovery Science*, Lecture Notes in Computer Science 2534, pages 332–339. Springer-Verlag, November 2002.
20. Y. Yamada, D. Ikeda, and S. Hirokawa. Frequency Analysis of Semi-structured Documents with Structural Similarity. In *Proceedings of the 2nd Forum on Information Technology (FIT2003)*, pages 59–60, September 2003. (in Japanese).