清 华 大 学

综 合 论 文 训 练

题目：大数据环境下信息抽取模板自动聚类与发现

系　　别：计算机科学与技术系

专　　业：计算机科学与技术

姓　　名：丘骏鹏

指导教师：朱小燕

辅导教师：郝宇

2013 年 6 月 12 日

关于学位论文使用授权的说明

本人完全了解清华大学有关保留、使用学位论文的规定，即：学校有权保留学位论文的复印件，允许该论文被查阅和借阅；学校可以公布该论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存该论文。

(涉密的学位论文在解密后应遵守此规定)

签　名：_____ 导师签名：_____ 日　期：_____

中文摘要

随着互联网的发展，研究人员可以获得越来越多的数据，我们已经进入了“大数据”的时代。在这样一个背景下，为了能使用计算机更高效地处理这些数据，我们需要从非结构化或者半结构化的数据中提取出我们关心的信息，并将其用结构化信息的方式存储下来。目前，互联网上的网页大多是通过模板动态生成的，为了从某些类似的网页中提取出结构化的信息，我们可以挖掘这类网页的共同点，找出这类网页的模板，然后用模板去抽取网页中的信息。

本文设计并实现了具有以下的功能的系统：对于海量的由各种网页组成的数据，先利用后缀树高效地找出每个网页中的重复记录并将其合并，然后通过聚类将不同模板生成的网页分开，再从每个类别中利用无监督方法抽取出对应的模板，利用这些模板去抽取我们需要的数据。

关键词：大数据；结构化数据；后缀树；聚类；模板；无监督学习

ABSTRACT

As the growth of the Internet, researchers now could obtain more and more data. We are in the era of “Big Data”. Under such circumstances, we need to extract the information we’re concerned with from the unstructured or semi-structured data and store it in the form of structured data so that computers can handle the data more effectively. currently, most of the web pages in the Internet are generated by templates. In order to extract structured data from these pages, we could dig into the common points of them and find out the template of a certain kind of web pages. Then we could make use of these templates to extract information from other web pages.

In this paper, we design and implement a system with following functions: for a large set of different web pages, it will first remove all the duplicate data records using a data structure called “suffix tree”. Then it will separate web pages which are generated by different templates through clustering, and for every cluster, it will generate a corresponding template by unsupervised learning. Finally, it could extract the information we need from other web pages using the generated templates.

Keywords: Big Data; structured data; suffix tree; clustering; template; unsupervised learning

目 录

第 1 章　引言

1.1　研究背景

1.1.1　大数据研究背景

随着互联网的快速发展，互联网上的信息呈现了爆发式的增长，互联网已经成为人们获取信息的一个主要渠道。举个例子，2011 年底，新浪微博注册用户数超过 3 亿，每日发微博量超过 1 亿[1]，在 2012 年底用户数更是突破了 5 亿[2]。到 2015 年, 将会有近 30 亿人在使用互联网，产生和共享的数据将达到 8ZB① [3]。随着我们可以获得的数据量的不断增加，人们的研究工作也受到了新的挑战。传统的数据处理手段正愈发显示出其局限性，如何有效对海量的数据进行处理，进而挖掘出我们所需要的内容逐渐成为一个重要的问题。近几年，“大数据”迅速成为计算机科学领域非常受关注研究方向。

Doug Laney 在 [4] 中，提出了“大数据”的 3 个特点：容量（volume）、速度（velocity）和多样性（variety）。容量是指数据的存储量非常大，通常在 TB，甚至 PB 级别；速度是指数据的产生速度很快；多样性是指产生的数据多种多样，没有一个固定的类型，大部分都是以非结构化和半结构化数据的形式存在。这些是我们在面对大数据时所需要解决的主要问题。

之前数据挖掘方面许多研究，更多地是关注如何在有限数据的情况下尽可能多地提取出准确的我们关心的信息。由于受到数据量的限制，很多数据中隐藏的模式和信息并不能被有效地发现。如今，海量的数据使得数据本身不再是我们研究中的瓶颈，我们关注的重点更多的在于如何从这些有大量重复冗余的数据中找到我们真正关心的那部分信息，将信息提取出来，组成结构化的信息，用于计算机的后续处理。

① 1 ZB = 10^{21}B

1.1.2 结构化数据简介

从结构上来看，数据可以分为非结构化数据，半结构化数据和结构化数据。结构化数据是指可通过明确的结构，如表或者树的格式，进行统一表示的数据。关系数据库和定义良好的 XML 就是存储结构化数据的两个典型例子。非结构化的数据没有统一的格式，比如各种各样的文本、图像、声音、视频等。半结构化的数据则有一定的结构，但结构并不固定，有些字段可能会扩充或者删除。目前人们日常所接触的大部分万维网上的信息，大部分都是通过 HTML 文档进行表示的，HTML 文档就是一种典型的半结构化数据，不同的文档在结构上可能有很大的变化。可以看出，非结构化数据和半结构化的数据更适用于人机界面的交互，而结构化数据则对机器更加友好。为了便于用计算机进行存储和后续处理，在用计算机处理各种各样的数据的时候，我们常常希望能将其他的非结构化或者半结构化的数据转化成结构化的数据进行表示。

这篇文章中，我们主要关心的对象是半结构化的 HTML 文档。为了更好地对 HTML 文档进行处理，我们需要将 HTML 文档中我们关心的信息提取出来，用结构化数据的方式（比如 XML）进行存储。例如，对于我们获得的博客数据，我们主要关心其中的标题，正文和评论的信息，那么可以建立一个 XML 文档，其中的字段都是固定的，每个文档对应一个 `document` 节点，`document` 节点下面有 `author`,`content` 和 `comment` 三个子节点。我们将每个博客的 HTML 中对应部分抽取出来，存储到该 XML 文档中。如图 1.1所示。

1.1.3 HTML 文档的模板

互联网上有成千上万的 HTML 文档，对于大部分的网站来说，不可能针对每一个网页单独写一个静态网页存储到服务器。实际上，我们在互联网上浏览的大部分 HTML 网页都是通过网站的后台程序动态生成的，只有极少量的还是通过静态 HTML 方式进行存储。

在 Web 开发领域，MVC(Model, View, Controller) 模式是目前最流行的开发方法。模型（Model）是对底层数据和业务的进行的封装；视图（View）负责用户界面的交互，包括给用户发送信息，接受用户输入等；控制器（Controller）则是系统的控制逻辑，对用户请求进行处理，用选择合适的视图用于显示模型返回的数据。如图 1.2所示[①] 。目前有很多基于 MVC 模式的 Web 开发框架，比如

① 来源：http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

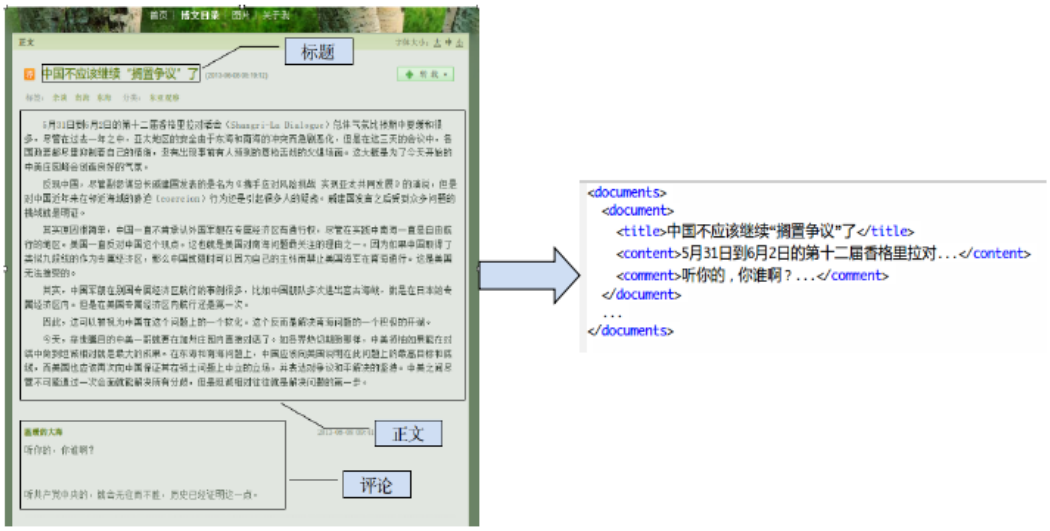


图 1.1　用 XML 存储博客的正文，标题和评论

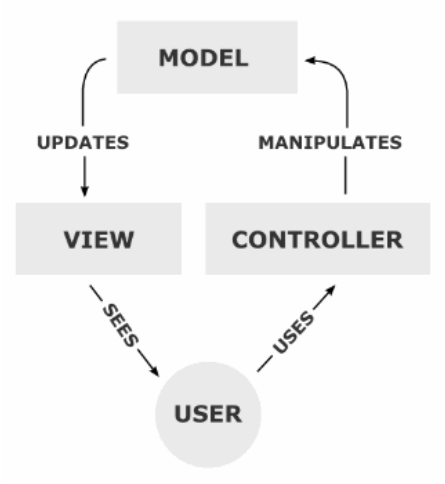


图 1.2　MVC 模式

```
<html>
  <body>
    <h1>{{ news.title }}</h1>
    <p>{{ news.content }}</p>
  </body>
</html>
```

图 1.3　Django 中的 HTML 模板示例

基于 Ruby 语言的 Ruby on Rails，基于 Python 语言的 Django 以及基于 Scala 语言的 Play! Framework 等等。这些框架用模型对底层的数据库进行包装，当用户请求一个 HTML 页面的时候，控制器负责从数据库中查询相应数据，然后在视图层选择合适的渲染模板，将对应的数据填充到模板中，从而生成目标 HTML 文档，将其返回给用户。图 1.3是一个简单的用 Django 开发网站时视图层的模板示例，其中 `news.title` 和 `news.content` 是从数据库中得到的查询结果。

```
<html>
  <body>
    <h1>{{ news.title }}</h1>
    <p>{{ news.content }}</p>
    <div>
      {% for comment in news.comments %}
        <li>{{ comment }}</li>
      {% endfor %}
    </div>
  </body>
</html>
```

图 1.4　使用了 for 的 Django 模板示例

我们注意到，大部分 HTML 网页都是通过这种查询底层数据库得到相应数据，然后使用模板进行渲染的方法生成的。对于大量的从同一个模板生成的网页来说，“模板”就是这些网页的“公共部分”。实际上，模板的定义要比这里所说的网页的“公共部分”要复杂一些，许多 Web 开发框架的模板引擎都支持一些简单的控制结构（如 if，for 等），如果生成模板的时候使用了这些控制逻辑，对应的模板也会比较复杂，如图 1.4所示。我们在 5中将会给出一个较为严格的定义，这里我们先使用这种直观的定义便于理解。如果我们能够从大量的由同一个模板生成的网页中将它们的模板抽取出来，那么我们就可以用抽取出来的模板对其他的由同一个模板生成的网页进行内容的抽取。简单来说，有了网页的模板以后，每个网页中非“模板”的部分即可以认为是通过查询后台数据库得到的数据动态生成的，而这些数据正是我们所需要提取出来的。

1.2　本文的主要工作

本文的主要工作是从已经抓取的大量的网页数据中，自动地对其中的网页进行聚类，得到不同的类别。对于其中的每一类，将其中可能的模板抽取出来，然后再利用这些模板去抽取新抓取的网页。由于我们的数据量比较大，因此需要设计一些高效的算法进行抽取；同时，海量数据也提供了较多的冗余性，使得其中一些隐藏的模式和信息可以更好地被挖掘出来，得到比较好的实验结果。

一个 HTML 文档由标签和标签所包含的内容所组成。如果仅考虑标签的话，可以根据标签和标签之间相互的嵌套关系定义一颗仅由标签所组成的树，我们将这样的一棵树称为 HTML 文档的结构特征。与之类似，HTML 文档中所包含的除标签以外的内容我们称之为 HTML 文档的内容特征。参考 1.1.3所述的网站

后台利用模板生成网页过程，我们认为模板实际上决定了生成的 HTML 文档结构特征中最主要的一部分（不是全部），而通过查询后台数据库中得到的结果动态生成的部分则主要决定了 HTML 文档的内容特征。按照这种理解，由同一模板生成的网页将会有较高的结构相似度，而不一定有很高的内容相似度——因为后台数据库中存储的数据本身没有很大的联系。因此，本文将重点放在了考察 HTML 文档的结构特征上，而对文档的内容特征不做太多关注。

互联网上有大量的网站，每个网站的网页都可能采用不同的模板生成。实际上，即使对于同一个网站，也可能有多个不同的模板。因此，我们需要先对这些网页数据进行聚类，使得同一个模板生成的网页位于同一类中，便于后续的模板提取。

在模板表示和提取方面，我们将通过寻找结构相似网页中共同的结构特征提取出模板，并利用提取出的模板对其他网页进行信息抽取。我们希望可以提取出类似于图 1.3中一样的模板，其中动态生成的部分（对于图 1.3来说，即 `{{}}` 中包含的 Python 表达式）我们用正则表达式 `.*` 代替。

1.3 相关工作

1.3.1 网页结构相似度计算

网页的结构特征相似度是我们进行聚类时的标准，因此快速准确地计算出网页的结构相似度是我们整个工作中非常重要的一部分。在这方面，已经有很多的相关的研究工作。

早期的工作主要集中在 HTML 文档解析成的 DOM Tree 上，主要的度量方法是树编辑距离（Tree Edit Distance）。Tai 和 Kuo-Chung 在 [5] 中提出了计算树编辑距离的方法，Reis 等在 [6] 中说明了如何利用树编辑距离的方法自动抽取网页新闻。此外，还有一些优化树编辑距离计算时间的尝试，包括 [7–9] 等。树编辑距离的主要缺点是算法非常复杂，计算复杂度很高，有些时间上的优化的算法又依赖于各种各样的限制条件。由于我们要处理的数据量较大，采用复杂度很高树编辑距离算法进行计算是不现实的。

Gottron 在 [10] 中介绍了几个基于 HTML 的标签（tag）集合的计算 HTML 文档结构相似度的方法。最简单的一种方法是计算两个 HTML 文档 D_i, D_j 的标

签集合 T_i, T_j 的 Jaccard 相似度，即

$$Sim(D_i, D_j) = \frac{|T_i \cap T_j|}{|T_i \cup T_j|}$$

这种方法在 HTML 文档上很难取得比较好的效果，因为 HTML 文档的标签集合是有限的，因此考察标签集合的 Jaccard 相似度很难体现不同的模板生成的 HTML 文档之间的区别。另一种简单的方法是标签矢量法，即将标签转化为矢量 V_i，每个标签出现次数 N_{t_i} 对应矢量 V_i 的一个分量，然后计算两个矢量的欧几里得距离，得到文档的相似度。

$$Sim(D_i, D_j) = \sqrt{\sum_{k=1}^{N}(v_k(D_i) - v_k(D_j))^2}$$

这种方法的缺点是仍然没有考虑标签之间的嵌套关系，只考虑了标签的数量关系，因此丢失了很多 HTML 文档结构上的信息。比较好的一种方式是通过某种树的遍历方法，得到树 T_i 一个标签序列 S_i，然后计算两个标签序列的最长公共子串（Longest Common Sequence），

由此得到两者的相似度。即

$$Sim(D_i, D_j) = \frac{LCS(S_i, S_j)}{max(|S_i|, |S_j|)}$$

还有一类计算相似度的方法是基于路径集合。路径（Path）指的是从根节点到某个目标节点的标签所组成的序列。Joshi 在 [11] 中提出了利用路径集合来衡量文档的结构相似度。Buttler 等人在 [12] 引入 shingle 技术，加上一些随机取样等优化手段，可以在常数时间内计算任意大小的文档之间的相似度。Kim 在 [13] 基于路径集合提出了一种以最小描述距离（Minimum Description Length）为优化目标、通过 Minhash 加快计算速度的计算方法。基于路径集合的方法将 HTML 的树形结构的相似度转化为序列集合的相似度，序列中标签的顺序仍然保留了原始 DOM Tree 的一些层次信息。这种方法与计算树编辑距离的方法相比，在不损失太多的结构信息的同时有效降低了复杂度，并且有多种优化的途径，是一种比较好的计算结构相似度的方法。

Flesca 在 [14] 中还提出了一种有趣的计算文档相似度的方法。他引入信号处理领域的快速傅里叶变换（FFT）用于处理文本。具体的方法是将文本的序列看成时序序列，利用快速傅里叶变换将其变换到频域，然后比较幅度的差别。

尽管找不到一个可以与之对应的合理直观的解释，同时 Buttler 在 [12] 中做的相关的实验也显示这种方法的效果不是很好，不过就方法本身来说，还是非常有启发意义的。

在 Buttler 等人在 [12] 的实验中，还有一个值得注意的结果。虽然树编辑距离看上去是衡量两个文档结构距离的最好标准，但是在某些实验中却显示树编辑距离算法在 HTML 文档的聚类效果上不如一些简单的近似算法，比如加权的标签算法和基于路径集合的近似算法，而且算法在运行时间上远远大于这些近似算法。

1.3.2 模板检测与提取

在网页模板的检测和提取方面，也有很多的相关的研究成果。典型的无监督的方法是 Arasu 等人的 [15]。这篇文章提出了一套模板推导引擎，通过比较不同网页间的相同点和不同点来自动生成网页模板。这种方法的特点是没有语义信息，需要人工后期指定相关的语义。

半监督的方法则往往同 bootstrapping 相结合。Calson 在 [16] 中，利用部分树对齐（partial tree alignment）的方式先对网页结构进行对齐，然后利用一些语义特征，对相似度进行打分，训练模型的时候采用了 bootstrapping，减小了标注的工作量。这种方法的缺点是算法运行速度较慢，不适用于数据量较大的情况。

还有一些工作并没有直接在 HTML 文档的 DOM Tree 的层次上进行，而是采用更符合人直观的网页视觉分块作为基础进行模板提取。Cai 等人在 [17] 中提出一种利用 HTML 标签的视觉属性信息来检测网页中的视觉分块的方法。这种方法的计算方法比较复杂，而且目前主流的网页都已采用 HTML+CSS 架构，原文中布局信息的计算也需要做较大的修改。

1.4 本章总结

本章最开始主要简单介绍了大数据的研究背景、结构化数据和 HTML 文档的模板。接着对本文的主要工作进行了概述，简单说明了我们将在模板聚类和抽取的过程中主要考虑 HTML 文档的结构信息。最后我们对网页结构相似度计算和网页模板提取方面已有的研究工作进行了总结和回顾。

接下来的几章将按以下方式组织：

第二章主要介绍系统的整体框架以及每个模块对应的设计，主要包括预处理模块、网页聚类模块和模板生成模块。

第三章主要介绍预处理模块中如何对 HTML 文档的重复记录进行检测。这一步主要利用到了后缀树的数据结构，我们将在这一章中详细介绍基于后缀树的重复记录的检测算法。

第四章主要介绍网页聚类模块的实现。我们以最长公共子串为基础计算网页的结构相似度，然后利用层次聚类算法进行聚类。

第五章主要介绍网页模板生成和内容的提取。我们将在之前的聚类结果和最长公共子串算法的基础上提出一种无监督的模板生成算法，用于内容的提取。

第六章介绍系统的具体实现细节以及实验的设定和结果。

第七章总结工作，对未来进行展望，提出改进和努力的方向。

第 2 章　系统框架

2.1　整体框架介绍

本文的主要工作在于搭建一个自动的网页模板抽取系统。系统主要分为三个模块：“预处理模块”，“网页聚类模块”和“网页模板生成和内容提取模块”。从输入的网页集合开始，依次经过以上三个模块，生成模板后用于提取新的网页中的信息，然后将提取的信息以 XML 的格式输出成格式化数据。

整体的框架示意如图 2.1所示，图 2.1包括了系统的训练和测试。其中，系统训练的输入是目前已经从互联网上所获得的网页集合，训练的输出为这些网页集合所对应的模板。系统的测试输入为新的网页，测试的输出为 XML 格式的数据。

以下将逐个介绍各个模块的总体设计。

2.2　预处理模块

预处理模块主要以下分为两个子模块：过滤无用网页和简化 HTML 文档。

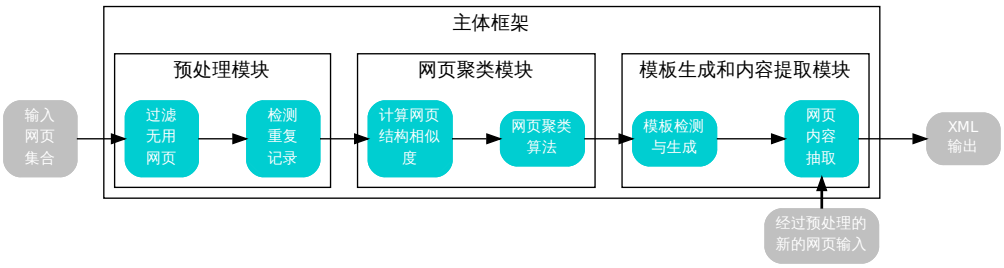


图 2.1　系统整体框架示意

2.2.1 过滤无用网页

我们获得的网页数据主要分为以下几种：目录页、详细页、404 及其他错误页。在我们的实验中，我们只关心详细页的集合，因此，需要首先过滤掉目录页和错误页等无用的网页。

这部分要求要有较快的运行速度，因此不需要设计非常复杂的算法，主要通过设置一些简单的规则去对原始的网页集合进行过滤。通过观察，我们主要采取了以下这些策略：

- 404 及其他类似的错误页可以通过一些关键字如 `404 Not Found` 或者 `Server Error` 等进行辨别。通常这些错误页的文档都很简单，长度很短，因此还可以通过设置长度阈值将其过滤；这样还可以将一些无用的太短的文档同时去除。
- 大部分目录页的 URL 和详细页的 URL 有明显区别。比如对于新浪博客来说，详细页的 URL 都以`.html` 结尾，而目录页则没有。例如以下两个 URL：

  `http://blog.sina.com.cn/u/1439351555`

  `http://blog.sina.com.cn/s/blog_55cac30301016yb1.html`

  第一个 URL 对应的是某个博主的文章目录，第二个 URL 则是该博主的某篇文章。因此只需要通过正则表达式匹配的方式即可将目录页过滤。其他网站也有类似这样的规则，我们只需要在实验前针对每个网站提前设置好规则进行过滤即可。由于每个网站每种特定的网页只需要指定一种规则，人工的工作量并不大。
- 为了实验的完备性和对一些特殊的网站进行处理，比如可能存在某些网站，目录页的 URL 和详细页的 URL 在模式没有太大的区别，或者人工设置规则的办法太麻烦了，我们的系统中也实现了通过网页的特征过滤详细页的机制。注意到目录页的作用主要是给用户提供进入各个详细页的入口，因此目录页的 HTML 文档的主要部分是由列表环境 `<ul>` 或 `<ol>` 中的列表项 `<li>` 及其包含的超链接标签 `<a>` 组成。图 2.2是简化后的百度新闻目录页 HTML 代码的一部分[①]。我们提取出网页中的所有 `<li>` 标签中的 `<a>` 标签，计算这些 `<a>` 标签的文本内容占网页所有文本内

① 来源：2013 年 6 月 9 日 15 时的 news.baidu.com 页面

```
<ul>
  <li><a>红会社监委今日讨论是否重启调查郭美美事件</a></li>
  <li><a>北京今日14时起堵车 中度以上拥堵将持续7小时</a></li>
  <li><a>湖北黄冈拒卖地反投8亿建免费公园 市长称不亏</a></li>
  <li><a>湖南株洲市档案局工会主席杀害局长后跳楼</a></li>
  <li><a>陕西神木集资大王被查 结婚时曾空运20辆林肯</a></li>
  <li><a>内地学生扎堆报京沪港高校 在校平均恋爱0.78次</a></li>
</ul>
```

图 2.2　简化的百度新闻目录页的一部分 HTML 代码

容的比重，超过一定的阈值则判定为目录页。由于这种方法比上述使用简单规则的办法要慢许多，因此在系统中我们将优先使用规则的方法对目录页进行过滤。

通过这些策略将目录页和错误页过滤完以后，我们可以得到网页的详细页集合，将其作为下一个子模块的输入。

2.2.2　简化 HTML 文档

我们将分多个步骤对 HTML 文档进行简化。

第一步是去除无用的标签。这些标签有以下特点：

1. 与网页模块无关。去除这些标签时，可以将标签本身及标签的内容一同去掉。比如 `<script>`、`<link>`、`<style>` 等。
2. 通常是深层次的 HTML 文档的节点，用于控制格式，在模板中意义不大。去除这些标签时只取出标签本身，保留标签所包含的内容。比如 `<br/>`、`<p>`、`<strong>`、`<em>` 等。
3. 我们在实验中不考虑非文本标签，因此将 `<img>`、`<audio>` 等直接去除。
4. 最后是去除一些在模板中变化很大的太复杂的标签，典型的是表格相关的一些标签，包括 `<table>`、`<th>`、`<tr>`、`<td>` 等。

接下来我们把每个 HTML 文档都解析成 DOM Tree。DOM Tree 的构建速度比较慢，在解析时需要将整个 HTML 同时载入内存，占用较多的内存资源，可能导致我们在需要处理的数据量较大时会出现一定的问题。去除这些无用标签可以加快 HTML 解析成 DOM Tree 的速度，减小内存占用，也减小了后续的模板聚类和模板检测中的噪音，是很有必要的。

第二步是对解析好的 DOM Tree 进行简化。由于在树形结构上做相关的操作较为复杂，我们首先将树形结构转化为更便于处理的序列形式。具体地，我们采用先序遍历的方式，将 DOM Tree 转化成一个标签序列，为了可以还原成原始的 DOM Tree，我们在遍历的同时在每个节点处保存了该节点的深度信息。

```
<html>
  <body>
    <div>
      <a></a>
      <p></p>
    </div>
    <div></div>
  </body>
</html>
```

图 2.3　一个简单的 HTML 文件

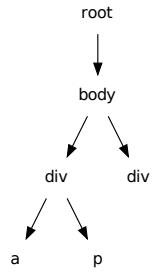


图 2.4　图 2.3对应的 DOM Tree

对于图 2.3所示的 HTML 文档，解析成的 DOM Tree 如图 2.4所示，如果每个标签采用标签名 + 深度的表示方法，则我们采用先序遍历方式可以得到该 DOM Tree 对应的序列为：`<html0><body1><div2><a3><p4><div2>`。先序遍历的优点在于可以保证每个子树的所有节点在序列中的位置是连续的，且根节点位于这段连续子序列的起始处。比如以第一个 `<div>` 标签为根的子树在遍历得到的序列中对应的序列为 `<div2><a3><p4>`，根节点 `<div2>` 位于子序列的起始位置。这些特点对于后续的处理有很大的帮助。

第三步是对 HTML 文档中重复出现的记录进行检测，去掉重复的多余的部分，从而用更简单的方式表示原文档。我们将 DOM Tree 的某个子树成为一个“记录”。由于 HTML 模板中动态生成的部分中可能包含大量重复的模式，比如图 1.4所示的 Django 模板，其中的 `news.comments` 可能包含不定数目的元素，在比较由这个模板生成的两个不同网页的时候，应该将其中数目可能不一样的 `<li>` 的结构视为一样，否则将对后续的处理造成较大的误差。在网页的生成中，这种重复模式是大量存在的，因此，检测网页中的重复记录并做相应的简化是预处理过程中最重要的一个部分，我们将在第 3章中详细介绍。

2.3　网页聚类模块

这个模块包含两个主要的子模块：计算网页结构相似度和实现网页聚类算法。

通过网页的预处理模块，我们将得到一个简化的由 DOM Tree 先序遍历得到的序列，其中不包含重复的记录。为了能够进行聚类，我们首先需要计算

每个文档之间的结构相似度。这里我们采用的是基于最长公共子串 (Longest Common Sequence) 的方法，简称 LCS。实际上，我们整个工作中与序列处理相关的工作很多都将基于 LCS，包括后面的模板生成和内容提取模块。我们将对每两个文档都计算一次 LCS，得到文档之间的结构相似度，将其用于后续的聚类。

在聚类子模块，有几个特点值得我们注意：

- 网页中的模板个数即最终聚类的个数事先是不知道的，因此我们不能采用一些类似于 K-means 等需要预先设定聚类的个数的算法。
- 网页数量很大，需要有较快的执行速度。
- 我们可以利用的信息只有网页文档两两之间的结构相似度。

我们将采用简单的层次聚类的方法，通过设置阈值的方法控制聚类的类别个数。这个模块的具体实现将在第 4章中详细介绍。

2.4 模板生成和内容提取模块

这个模块是本次实验中最主要的部分，包括两个子模块：模板生成和内容提取。

在模板生成子模块中，我们将从之前聚好的类中提取出每个类对应的模板。模板的提取将仍然基于 DOM Tree 先序遍历得到的序列和 LCS，并利用无监督的方法来进行构建。这个子模块输出的模板将作为整个系统的训练模块的输出。

为了能够从网页中提取出我们所需要的部分，我们还需要人工对模板中的某些部分指定语义，比如对于新闻来说，模板中的哪些部分将对应新闻标题、正文、评论等。由于模板的数量不会很多，这部分的工作量不会很大。

内容提取子模块将利用之前训练得到的聚类信息和模板，选择合适的模板从新的网页中提取出我们需要的信息，并将这个信息以 XML 格式输出成结构化的数据。

这个模块我们将在第 5章中进行详细介绍。

2.5 本章小结

本章首先简单介绍了系统的整体框架，然后分别对每个模块的总体设计进行了说明。其中我们重点介绍了预处理模块中除了重复记录检测子模块以外的部分，其他概述的部分，如重复记录的检测，网页的聚类，模板生成和内容提取等，我们将在后续章节中一一详细介绍。

第 3 章　基于后缀树的重复记录检测

3.1　后缀树简介

后缀树 (Suffix Tree) 是一种可以快速高效实现许多字符串操作的数据结构。Weiner 最早在 [18] 中提出了后缀树的概念，Ukkonen 在 [19] 中提出了一个 $O(n)$ 时间复杂度的在线构造算法。

后缀树实际上是前缀树（Trie 树）的一种特殊形式。后缀树的每一条边都代表着一个序列, 从根节点到后缀树叶子节点的每条路径都对应着原序列的一个后缀。也就是说，后缀树其实就是由序列所有后缀所组成的前缀树。举个例子，对于字符串 `APPLE`，其所有的后缀为：

```
APPLE
PPLE
PLE
LE
E
```

对应的后缀树如图 3.1所示。从图 3.1中可以看到，共有 5 条从根节点到叶子节点的路径，每一条路径都对应于一条后缀。

我们可以总结出序列 S 的后缀树有以下几个特点：

1. 所有的边都对应 `S` 的长度至少为一的非空的子序列
2. 从根到叶子节点的路径序列和序列 S 的后缀一一对应
3. 所有的内部节点（根节点不一定）都有至少两个子节点

3.2　Ukkonen 后缀树构建算法

最简单的后缀树算法是先取出序列所有的后缀，然后按照前缀树的构造方法构造序列的后缀树。这种方法虽然简单，但是复杂度较高，不适用于数据量较大的情况。为了高效地实现重复记录检测的模块，我们的系统实现了

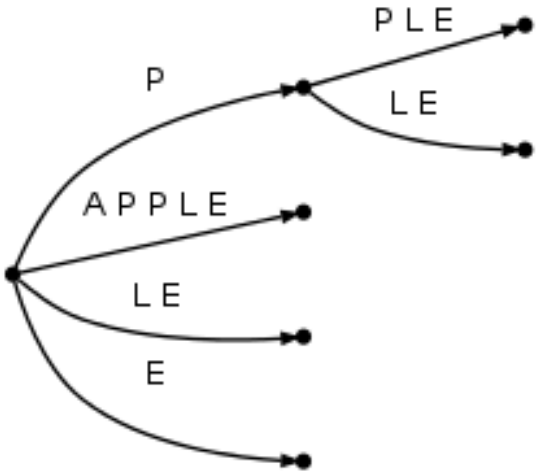


图 3.1　APPLE 对应的后缀树

Ukkonen 在 [19] 中提出的时间复杂度为 $O(n)$ 的后缀树在线构建算法。以下将以字符串 BANANA 为例，简单介绍这种在线算法的构造过程。由于这个算法比较复杂，以下的这个例子并不能涵盖算法的所有细节。

首先我们引入 3 个状态变量描述某一个时刻对应的后缀树的构造状态。

第一个状态变量是 #，表示序列中即将扫描的下一个元素的位置。初始值为 0，表示第一个元素，在以后的每轮迭代中，# 的值将加 1，当 # 的值和序列的长度相同时迭代终止。

第二个是 activePoint，用于表示下一次要在后缀树插入新的点时的位置，为了确定这样一个唯一的位置，我们需要一个三元组 (activeNode, activeEdge, activeLength) 表示 activePoint，这三个变量分别表示当前后缀树的某个节点，该节点的某条边以及这条边上第几个元素。初始值为 (root, ", 0)，表示初始的插入操作都在根节点 root 上进行。下面以图 3.1中的后缀树为例简要说明：若用 root 表示根节点，则当 activePoint 为 (root, A, 3) 时意味着下一次插入新的点的位置应该是从根节点 root 的以字母 A 开头的边的第 3 个字符后面，即 root 的 APPLE 对应的边的第二个 P 字母后面的位置。

第三个状态变量是 remainder，表示当前还需要往后缀树中插入几次后缀。初始值为 1。

对于我们的输入 BANANA 来说，最后一个字母是 A，这个字母在字符串之前的某个位置中出现过（BANANA 共有 3 个字母 A），因此需要在输入串的末尾

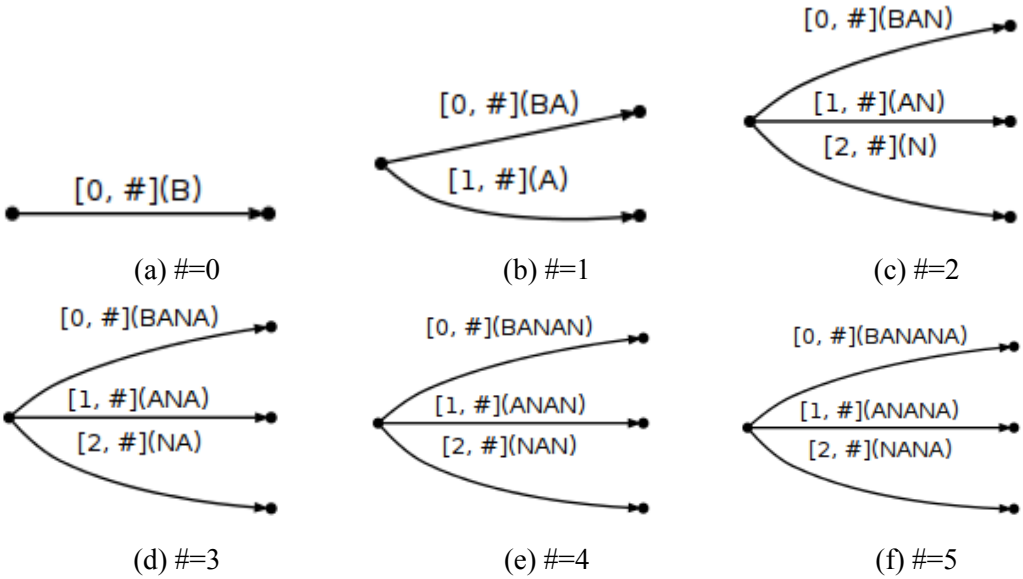


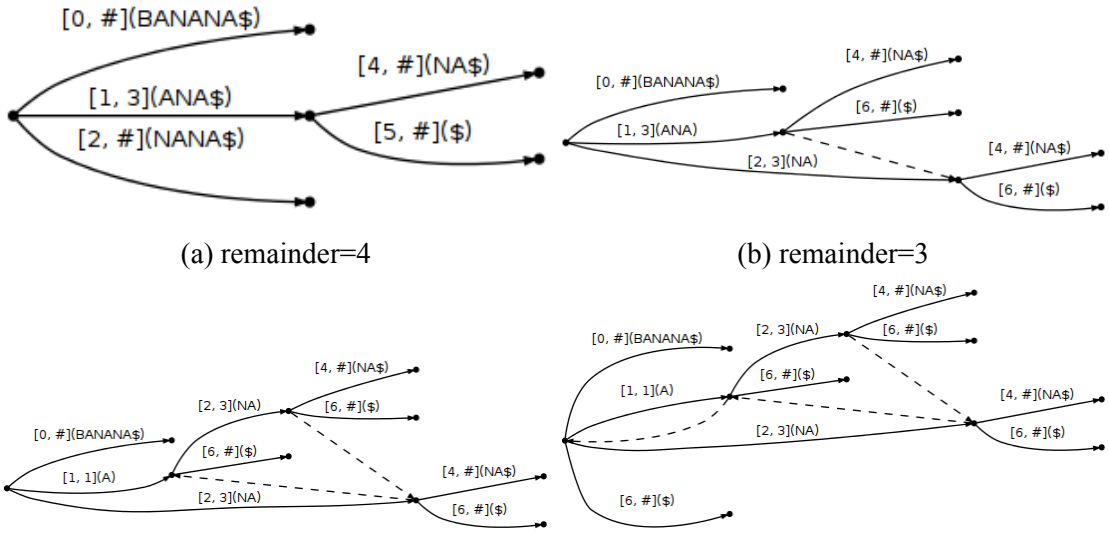
图 3.2　前六轮迭代（# 从 0 到 5）中的后缀树变化

增加一个终结符，以保证输入的最后一个字符没有在之前出现过，我们这里采用 $。因此，原始的输入 BANANA 就变成了 BANANA$

一开始的时候，我们只有一个根节点，# 为 0，插入后缀 B，变成图 3.2(a)所示的图；下一轮迭代 # 值增加到 1，新的后缀 A 在原来的边中没有出现，插入一条新的边，得到图 3.2(b)；接下来 # 增加到 2，和前一步一样，新的后缀 N 没有在原来的边中出现过，插入一条新的边，得到图 3.2(c)。目前为止，activePoint 和 remainder 都没发生变化。

接下来 # 加 1，将导致需要插入一条新的后缀 A，但是我们发现 A 已经在之前的边中出现过，因此这时我们不插入新的边，而将 activePoint 更新为 (root,A,1)，remainder 加 1，此时后缀树如图 3.2(d)所示，边数没有发生变化，但是状态变量发生了改变。下一步 # 再加 1，由于此时 remainder 为 2，因此我们需要将新的后缀 N 往后缀树中插入两次。从 activePoint 出发，我们发现下一个字母正好是 N，因此我们不再试图往后缀树中插入 N 了，而是将 activePoint 和 remainder 分别更新为 (root,A,2) 和 3。下一轮迭代也是类似，在插入最后一个 $ 之前，后缀树如图 3.2(f)所示，activePoint 和 remainder 的值分别为 (root, A, 3) 和 4。

最后一轮迭代，我们需要插入终止符 $。从此时的 activePoint 出发，发现下一个字母和和 $ 并不匹配，此时我们需要往后缀树中插入 $ 字符，方法



(a) remainder=4　　(b) remainder=3

(c) remainder=2　　(d) remainder=1

图 3.3　最后一轮迭代（#=6）时的后缀树

是在 activePoint 对应的位置生成一个新的内部节点，并在新生成的节点的地方插入一条边，表示 $，如图 3.3所示。我们需要更新 activePoint 的值：将 activeNode 移到根节点处（在这个例子中实际上没有移动），对应的边是以原来的 activeEdge 的第 2 个字母开头的边，即 N，activeLength 减 1。remainder 的值也减 1，变为 3，也就是我们还需要插入 3 次 $。接下来的两次插入和我们第一次插入时的情况一样，会新生成两个内部节点；最后一次插入则和一开始的情况一样，我们直接在根节点处插入 $。最后一轮迭代的过程如图 3.3所示，最后生成的完整的后缀树如图 3.3(d)所示。

在最后的图 3.3(d)中，我们看到还有一些连接几个内部节点之间的有向的虚线，我们之前没有提到，这是后缀链（Suffix Link）。我们例子中的后缀链都是在最后一轮迭代插入 $ 时生成的。关于后缀链生成的简单的规则是：在同一轮迭代中，如果在某一步中某个节点处发生了一次插入，则要从这一轮迭代之前生成内部节点（如果有的话）引一条后缀链指向该节点。后缀链的作用是如果在某个带有后缀链的节点发生了插入，则在更新 activePoint 时把其中的 activeNode 更新为沿着后缀链连接的那个节点，而不是根节点。后缀链不影响前述的其他规则。限于篇幅，我们不再具体举例说明后缀链的使用方法。

3.3 重复记录检测算法

3.3.1 后缀树检测重复序列的基本算法

依照 3.2所述的后缀树构造算法，我们给每一个 DOM Tree 的先序遍历序列构造一个后缀树，然后根据后缀树检测其中重复的记录。

根据后缀树构造的特点，任意一条从根节点到内部节点的路径组成的序列都是原序列中重复出现的子序列，且重复的次数是以该内部节点为根的子树的叶子节点个数。在图 3.3(d)中，A,ANA,NA 都是 BANANA 的重复子串，分别重复出现了 3 次，2 次和 2 次。这种方法有以下几个问题：

- 不同的重复子串之间可能有包含或者相交的关系。在之前的例子中，ANA 包含了 NA 和 A。
- 同一个重复子串在原序列上有交集。比如 ANA，对应到原序列上的两个子序列有交集 A。

直接采用上面的方法查找后缀树中的重复记录是不可行的。我们从 DOM Tree 得到先序遍历序列和普通的序列不一样，里面是包含有原始的树的结构信息的。我们需要检测的重复记录必须位于同一个子树上，有公共的父亲。这个特点对应到序列上，必须要求：

- 重复序列不能横跨两个子树
- 重复序列必须有公共的父亲

此外，我们还要求这个重复序列必须尽可能地长。

3.3.2 改进的检测算法

因此，我们需要进一步对原来的重复序列查找算法进行改进。算法 1和2给出了我们查找所有符合我们要求的重复子序列的算法。

根据算法 1和 2得到的重复子序列，可以保证不会横跨两个子树，且尽可能地长，但还有几点需要说明。

第一，根据算法 1和 2得到的在原序列中重复出现的子树的序列，不一定有相同的父亲节点，因此不能直接根据这个结果去掉重复记录，还需要做进一步的处理。

第二，算法 2中的代码 addToResults($prefix + seq, results$) 将新的序列加入到结果集合中。实际在加入的时候，我们需要先将新的序列和结果集合中已有

算法 1 从根节点出发，找出所有的重复子序列

输入: 已经构建好的后缀树，根为 *root*

输出: 该后缀树中所有的重复子序列

```
1: // 从节点出发，寻找所有的重复子序列
2: for edge ← root.edges if edge.endNode.isNotLeaf do
3:     // 取后缀树根节点的每条边的第一个元素作为每个子树的根节点
4:     subTreeRoot := edge.firstElement
5:     // 查找以该节点为根的所有重复子序列
6:     findAllRepetitions(root, nil, subTreeRoot)
7: end for
```

的序列进行对比。如果结果集合中的某个序列被新的序列所包含，则我们用新的序列代替原有的序列。这个操作可以在 $O(1)$ 的时间内完成。因为在后缀树中，如果两个子序列有包含关系，即 S_1 包含 S_2，必然有 S_2 为 S_1 的后缀，这可以通过比较 S_1 和 S_2 最后一个元素的下标是否相等来判断。

3.3.3 合并重复记录

接下来我们将对找出来的所有可能的重复序列进行合并。根据先序遍历的特点，子树的根节点总是出现在该子树对应的序列的起始处。因此，对于我们找到的每一个重复的子序列，我们总是可以通过向前搜索找到其父节点。接下来把所有的序列按照其父亲节点进行分类，然后遍历每一个父节点所对应的全部重复序列 $S_1, S_2, ..., S_n$，若存在一个序列集合 $S_{i_1}, S_{i_2}, ..., S_{i_k}, i_1 < i_2 < ... < i_k, k > 1$，其中的每个序列都相等，则只保留 S_{i_1}，将其余全部去除。最后我们将在保留下来的 S_{i_1} 中设定标志位，表明该序列在模板中是可以多次出现的。

我们实际的合并算法要比上述的稍复杂一些，因为有很多边界情况需要特殊处理。比如，算法 1和2虽然去除了序列横跨两个子树和序列互相包含的问题，但是对于序列相交的情况，并没有做相应的处理。对于以下的某个 HTML 标签序列片段（每个标签由标签名 + 深度组成）：

```
...<div3><div4><div3><div4><a5><img5><div4><a5><img5>...
```

按照之前的算法，我们可以找到的符合条件的重复序列包括 <div3><div4> 和 <div4><a5><img5>。但是这两个序列实际上有交集：<div4>。我们采取的

算法 2 找出后缀树中经过某个内部节点的所有可能的重复子序列

输入: 一个内部节点 *node*，当前已经找到的重复序列 $prefix$，要找的子树的根节点 *subTreeRoot*

输出: 所有经过该内部节点的符合要求的重复序列

```
1:  function findAllRepetitions(node, prefix, subTreeRoot)
2:      // 定义一个空集合
3:      results := Collection.empty
4:      // 对于该内部节点的每一条不连接叶子节点的边
5:      for edge ← node.edges if edge.endNode.isNotLeaf do
6:          // 依次取出该条边上元素的根节点可能为 root 的点
7:          seq := edge.takeWhile(element inSubTreeOf subTreeRoot)
8:          if seq.length == edge.length then
9:              // 遍历完了该条边上所有元素，
10:             // 则取该条边连接的下一个内部节点进行递归查找
11:             findAllRepetitions(edge.endNode, prefix + seq, subTreeRoot)
12:         else
13:             // 否则，将当前得到的序列加入到结果集合中
14:             addToResults(prefix + seq, results)
15:         end if
16:     end for
17:     return results
18: end function
19:
```

策略是只取更深的重复序列。在这个例子中，我们保留 <div4><a5><img5>，将第二个 <div3><div4> 序列从重复序列集合中去掉。

其他还有一些边界条件，这里不再赘述了。

3.4 本章小结

本章主要对预处理模块中的重复记录检测子模块进行了详细描述。先简单介绍了后缀树的定义和性质，然后举了一个具体的例子说明如何在 $O(n)$ 时间内在线构建出一个序列的后缀树，最后对利用后缀树进行重复记录检测的算法做了详细的介绍和说明。

这个子模块是预处理模块中最重要的一个。如果我们不对网页中重复的记录进行处理，这些不定个数的重复记录将成为后续的聚类和模板提取过程中很大的噪音，大大减小聚类和模板提取的精度。由于我们实现并使用了后缀树这种高效的数据结构，使得我们的算法可以很快找出所有符合条件的重复序列，并将它们正确地进行合并。

第 4 章　网页结构相似度计算与聚类

4.1　基于最长公共子序列的网页距离计算

我们将 DOM Tree 的结构相似度转化成经过预处理模块得到的新的序列的相似度。这里，我们将采用经典的最长公共子序列的方法计算两个序列的相似度。

假设经过预处理模块，文档 D_1, D_2 分别被转化为序列 S_1, S_2，长度分别为 len_1, len_2，定义一个大小为 $(len_1+1)*len(_2+1)$ 的二维表 T，表的某个位置 $t(i+1, j+1), i, j >= 0$ 表示 S_1 的前 i 个元素组成的子序列和 S_2 前 j 个元素组成的子序列的最长公共子序列长度。T 的第一行和第一列的所有元素初始为 0。根据 T 的定义，我们只要计算出 T 中 $t(len_1, len_2)$ 位置的值即可得到 S_1 和 S_2 的最长公共子序列长度。

计算表格 T 我们有以下的递推公式：

$$t(i)(j) = \begin{cases} 0 & i = 0,\ j = 0 \\ t(i-1)(j-1)+1 & i,\ j > 0, x_i = y_j \\ \max(t(i)(j-1), t(i-1)(j)) & i, j > 0,\ x_i \neq y_j \end{cases} \tag{4-1}$$

x_i 和 y_j 分别表示 S_1 的第 i 个元素和 S_2 的第 j 个元素。在以上递推过程中，我们还可以同时记录每个位置的值是由哪个位置的值计算得到的，这样就可以通过回溯得到两个序列的最长公共子序列了，记为 $lcs(S_1, S_2)$。

得到了最长公共子序列的长度 $|lcs(S_1, S_2)|$ 以后，文档 D_1 和 D_2 的相似度定义为：

$$Sim(D_1, D_2) = \frac{|lcs(S_1, S_2)|}{\max(|S_1|, |S_2|)}$$

4.2 算法优化与改进

4.2.1 优化和改进的动机

前面给出的最长公共子序列的算法非常简单，然而，为了可以进行聚类，我们至少需要对文档集合中每两个文档之间计算一次结构相似度。由于我们实验的数据量很大，即使进行离线处理，如果不做优化的话，时间上也无法承受。

假设我们目前实现的计算最长公共子序列的算法运行一次平均需要 $0.001s$，文档集合大小为 60000，在单线程的情况下，计算一次全部文档两两之间结构相似度的时间大约为：

$$\frac{60000^2}{2 * 3600} * 0.001 = 500h$$

$500h$ 大约是 21 天左右，对于我们的实验来说，这个时间是不可承受的。

因为最长公共子序列的计算也是我们后续模板抽取工作的一个基础，除了时间以外，我们也还需要对其他方面进行优化。以下我们将从 3 个不同的方面进行论述：空间，时间和计算的方式。

4.2.2 空间优化

原始的计算方式空间复杂度为 $O(mn)$，在文档数很多的情况下，可能会造成很大的内存占用，因此空间上的优化是有必要的。空间优化的方法比较简单。观察公式 (4-1)，我们可以看到，每个位置的值实际上只依赖于上一行计算出的结果和这一行之前的计算结果，因此我们不需要 $(len_1 + 1) * (len_2 + 1)$ 大小的二维表，只需要保存两个一维数组即可。这样我们把算法的空间复杂度降到了 $O(n)$。

4.2.3 时间优化

时间上的优化是我们这部分工作的重点。我们采用的策略是并行计算，具体实现中用到的并行计算编程模型为 Actor。

Actor 模型是并行计算中常见的一种模型，早在 1973 年 Hewitt 就在 [20] 中就提出了 Actor 的概念，到目前已经有 40 年左右的历史了。现代的很多编程语言都内建了对 Actor 的支持，比较有名的有 Erlang 和 Scala 等。

Actor 模型的基本哲学和面向对象编程的思想非常类似，即一切都是 Actor。然而，和面向对象最大的不同在于：面向对象的程序是串行执行的，而 Actor

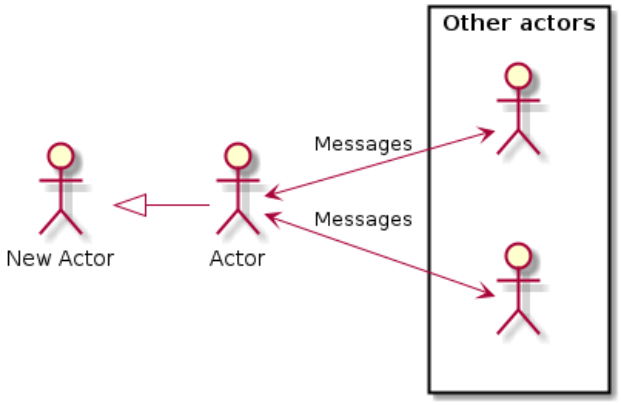


图 4.1　Actor 模型简单示意图

模型本质却是并行的。每个 Actor 都是一个独立的计算单元，可以接受并处理其他 Actor 发来的消息，也可以向其他的 Actor 发送消息，还能创建一些新的 Actor。以上的这些行为都可以并行地进行，没有固定的顺序。Actor 之间只能通过异步消息来进行信息传递，相互之间不共享状态，也不会在因为等待消息而堵塞，避免了由于同步和全局共享状态而导致的锁和其他问题。Actor 模型的简单示意图如图 4.1所示。

我们的系统中在实现中采用了基于 Scala 语言的 Actor 库 Akka[①]。接下来我们需要对计算任务进行分割，以便使用 Actor 模型来并行地进行计算。

对于一个文档集合，我们将计算其中两两文档之间的结构相似度的任务映射到几何平面上，相当于计算一个正方形按对角线分割的一个三角形区域，三角形区域的某个点 (i, j) 表示文档 D_i 和 D_j 之间的结构相似度。我们用等距的水平和垂直线分割这个三角形区域，把整个任务分成许多子任务，如图 4.2所示。计算的时候，建立一个主 Actor 和一个监听 Actor，主 Actor 负责根据配置创造新的 Actor，并使用调度器（我们使用的是简单的 RoundRobin 调度算法[②]）来向每个 Actor 动态分配任务。当主 Actor 创建的 Actor 都已经计算完毕，并且没有剩下的计算任务可以分配的时候，主 Actor 负责向监听 Actor 发送消息，告知任务完成，监听 Actor 则负责将整个系统关闭。这样一个计算的模式我们后面还会用到，因此我们将其做了封装，隐藏了底层的 Actor 模型的实现细节，便于后续的使用。

① 网址为：http://akka.io
② 这是一种经典的调度算法，可参见 http://en.wikipedia.org/wiki/Round-robin_scheduling

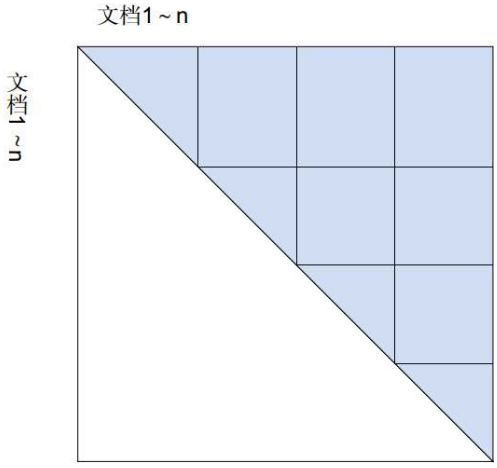


图 4.2　任务分割示意图

由于使用了并行的计算方式，在有较多计算资源的计算机上，计算网页结构相似度的模块可以较快地完成。

4.2.4　计算方式优化

在原始的实现中，我们忽略了在先序遍历 DOM Tree 的时候保存下来的节点的深度信息。实际上，不同深度的节点对模板的影响是不一样的，离根节点越近，即深度越小的节点，其成为模板的一部分的可能性就越大；反之，其成为模板一部分的概率就越小。因此，我们可以将最长公共子序列的计算方式进行扩充，根据节点的深度信息给表格 T 的每个点进行加权，设加权函数为 $f(depth)$，则修改后的算法的递推式变为：

$$t(i)(j)=\begin{cases}0 & i=0,\ j=0\\ t(i-1)(j-1)+f(x_i.depth) & i,\ j>0, x_i=y_j\\ \max(t(i)(j-1), t(i-1)(j)) & i,j>0,\ x_i\neq y_j\end{cases} \tag{4-2}$$

这样做的好处是将一些深度很大的节点的权重减小，使得算法更倾向于将深度更小的节点进行匹配，因此可以得到更好的匹配效果。修改后的算法得到的 $t(len_1)(len_2)$ 的值并不是简单意义上最长公共子序列的长度，我们将其记为 $|elcs(S_1,S_2)|$，相应地，我们需要修改用于计算文档 D_1 和 D_2 的相似度的公式：

$$Sim(D_1,D_2)=\frac{|elcs(S_1,S_2)|}{\max(\sum_{n\in S_1} f(n.depth), \sum_{n\in S_2} f(n.depth))}$$

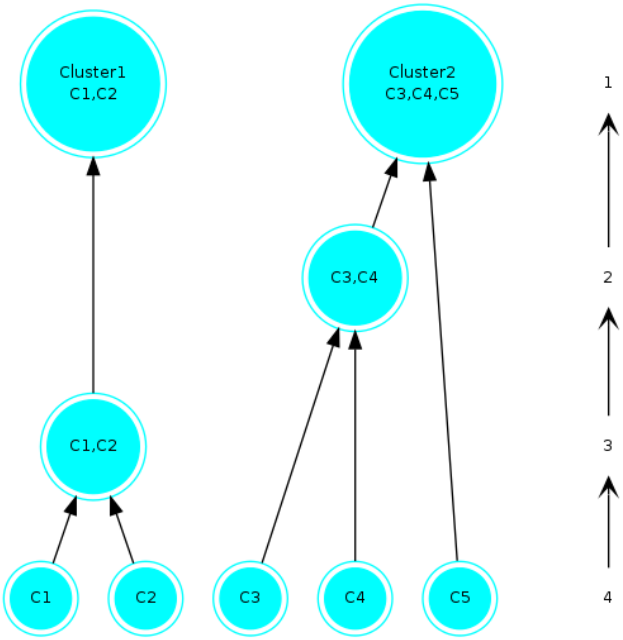


图 4.3　简单层次聚类方法图示

4.3　聚类算法实现

聚类的目的是为了将有明显差异的模板生成的网页分开，因此，我们可以认为需要分开的网页的结构相似度是比较低的。同时，在聚类之前我们并不清楚这些网页是由几种不同模板生成的，因此不能采用需要预先设置类别个数的聚类方法，比如 $K-means$。在我们的系统中，为了实现高效的自动的聚类，我们采用了一个简单的自底向上的凝聚层次聚类算法。

算法本身很简单：初始时，让每个文档实例都单独为一类，之后不断迭代，每次选择距离最近的两个类合并，直到任意两个类的距离都大于阈值时程序退出。这个聚类算法的结束条件由阈值决定，无需实现设定类的个数。具体的算法实现如伪代码 3所示。这里说明一下类中心点的更新方法：由于我们只有文档之间的结构相似度作为聚类的标准，每个文档实例本身没有合适的可度量的表示方法，不能求平均，因此类似 $K-medoids$，我们选择到类中所有点距离之和最短的点作为类的中心点。

聚类过程可以用图 4.3简单表示。这个例子中一开始有 5 个类，每一步选择最近的两个类进行一次凝聚，最后聚成两个类。

算法 3 简单的自底向上的凝聚层次聚类算法

输入: 所有的文档实例 *instances*，已经存储下来的计算好的距离 *distances* 和聚类的阈值 *threshold*

输出: 聚类后的类的集合 *clusters*

```
1:  function aggloClustering(instances, distances, threshold)
2:      // 初始化类的集合为空
3:      clusters := Collection.empty
4:      // 每一个文档实例单独看成一个聚类，加入到类集合中
5:      for inst ← instances do
6:          c := new Cluster(inst)
7:          clusters.add(c)
8:      end for
9:      // 不停迭代，直到最接近的两个聚类的距离大于设定的阈值为止
10:     while true do
11:         // 从全部聚类中取出两个距离最近类
12:         (c1, c2) := getMinDistanceClusterPair(clusters)
13:         // 如果距离已经大于预先设定的阈值，退出
14:         if distances(c1, c2) > threshold then
15:             break;
16:         end if
17:         // 合并两个类，更新类中心，更新类集合
18:         c_merge := mergeCluster(c1, c2)
19:         clusters.remove(c1); clusters.remove(c2);
20:         clusters.add(c_merge)
21:     end while
22:     // 返回聚类结果
23:     return clusters
24: end function
```

4.4 本章总结

这一章我们先介绍了传统的最长公共子序列的计算方法，然后分别介绍了如何对原有的算法进行优化，包括空间、时间和计算方式上的优化。在时间优化的部分，我们引入了 Actor 作为我们并行计算的模型，并在系统中用 Actor 模型简单实现了我们文档结构相似度计算的模块。最后我们介绍了我们采用的凝聚层次聚类算法。通过设置合适的阈值，我们可以通过聚类算法将差异很大的模板生成的网页成功分开，之后这些聚类将作为下一个模块的输入，进行模板的提取。

第 5 章　模板生成和内容提取

5.1　模板定义

我们在 1.1.3中介绍了模板直观上的定义：不同网页公共的部分。我们期望从目前得到的大量的由同一模板生成的网页集合中，找出那些反复出现的网页结构，将其作为这些网页的模板。

这里我们将首先形式化定义模板的组成。模板的基本元素是模板节点（Template Node），每个节点有两种形式：

1. 单个不重复的 HTML 标签，即 $< tag >$
2. 由一个或多个 HTML 标签组成的序列，这些序列可以出现一次或多次，即 $(\sum_{i=1}^{N} < tag_i >)+$，其中 $N \geq 1$

可以看到模板节点与经过预处理去除重复记录后的序列节点是对应的。第一种模板节点对应没有合并的序列节点，第二种模板节点对应着多个重复记录合并后的序列节点，由于是一个“记录”，因此可以由一个或多个标签组成，并且可以重复出现。

模板节点的序列可以组成必选节点（Essential Node）或可选节点（Optional Node）。必选节点对应着由模板节点组成的一个序列，若模板节点用 tn_i 表示，则必选节点 EN 可以表示为：

$$EN = tn_1 tn_2 ... tn_n$$

可选节点 ON 则同时对应多个序列，每个序列由不同的模板节点组成，同时每个序列还对应着一个出现概率 p，即：

$$ON = tn_{11} tn_{12} ... tn_{1n_1}, p_1 | tn_{21} tn_{22} ... tn_{2n_2}, p_2 | ... | tn_{k1} tn_{k2} ... tn_{kn_k}, p_k$$

我们规定模板中必选节点和可选节点必须间隔出现，一个模板（Template）Tp 可以定义为这样一个序列：

$$Tp = [ON_0] EN_1 ON_1 EN_2 ON_2 ..... EN_n [ON_n]$$

其中第一个可选节点 ON_0 和最后一个可选节点 ON_n 都不是必需的。

5.2 模板生成

通过聚类，我们已经得到了一些由同一种模板生成的网页集合，接下来我们将对每个聚类单独进行处理，生成对应的模板。设当前我们处理的聚类为 c，聚类的中心点是 p_{center}，类中的点 p_i 对应的序列为 s_i。

根据定义，我们需要生成一个必选节点和可选节点交替出现的序列。我们的方法是：先生成全部的必选节点，然后在中间插入可选节点。

必选节点，顾名思义就是在该模板生成的网页中，必选节点中包含的所有模板节点都应该出现。因此，我们需要求出在 c 中每个序列都出现的那些标签序列 s_{common}。做法是从聚类的中心点 p_{center} 出发，将其对应的序列 s_{center} 作为 s_{common} 的初始值，依次与 c 中的每一个序列求一次最长公共子序列 lcs，并将 s_{common} 的值更新为 lcs。迭代结束时得到的标签序列就是组成必选节点的所有的模板节点。算法如伪代码 4所示。

算法 4 得到组成必选节点的所有模板节点

输入： 输入聚类 c

输出： 得到 c 中所有文档共有的序列 s_{common}

```
1:  function findCommonSequence(c)
2:      // 初始化为中心点所对应的序列
3:      s_common := c.s_center
4:      // 依次求最长公共子序列
5:      for s_i ← c.sequences do
6:          lcs := getLCS(s_i, s_common)
7:          s_common := lcs
8:      end for
9:      return s_common
10: end function
```

以上得到的序列只是所有网页结构的公共部分，每个由这个模板生成的网页都应该有这个结构，然后这并不是模板的全部组成。我们考虑到很多后台模

```
{% if news.has_subtitle %}
  <h1>news.title</h1>
  <h2>news.sub_title</h2>
{% else %}
  <h1>news.title</h1>
{% endif %}
```

图 5.1　Django 带条件判断的模板

板生成网页时可能会使用一些条件分支控制结构，如 if，因此仅仅由上述的序列组成模板是不够的。仍然以 Django 为例，有些网页的标题可能有副标题，而有些可能没有，在 Django 框架的模板语言中，可以写成如图 5.1所示的模板片段。根据这个模板的定义，<h2> 标签并不一定会在全部的由该模板生成的网页中出现，但我们在模板生成的时候必须考虑到这种情况，否则有些内容比如 <h2> 对应的“副标题”就没法正确提取了。为此，我们引入了可选节点。因为实际的模板中可能会有多种不同的条件分支，所以每个可选节点对应多个不同的模板节点的序列，这些序列不一定在每个网页中都出现，每个序列都对应着一定的出现概率。

以下我们举个简化的例子来说明构造可选节点的基本思想。设有 3 个序列，分别为：

$$
\begin{aligned}
s_1 &= aorzbcdlxe \\
s_2 &= athubeatcdlxe \\
s_3 &= athubeatcdpkue
\end{aligned}
$$

首先，根据算法 4，得到 s_{common} 为 $abcde$。然后，我们将每个序列同 s_{common} 进行对齐，得到

$$
\begin{array}{rcccccccc}
s_1 & : & \mathbf{a} & orz & \mathbf{b} & & \mathbf{cd} & lx & \mathbf{e} \\
s_2 & : & \mathbf{a} & thu & \mathbf{b} & eat & \mathbf{cd} & lx & \mathbf{e} \\
s_3 & : & \mathbf{a} & thu & \mathbf{b} & eat & \mathbf{cd} & pku & \mathbf{e} \\
s_{common} & : & \mathbf{a} & & \mathbf{b} & & \mathbf{cd} & & \mathbf{e}
\end{array}
$$

我们利用每个序列同 s_{common} 对齐的结果，对每个序列进行分割，我们用 $s_i[t_1, t_2]$ 表示序列 s_i 在对齐的标签 t_1, t_2 之间的部分，如 $s_3[a, b]$ 表示序列

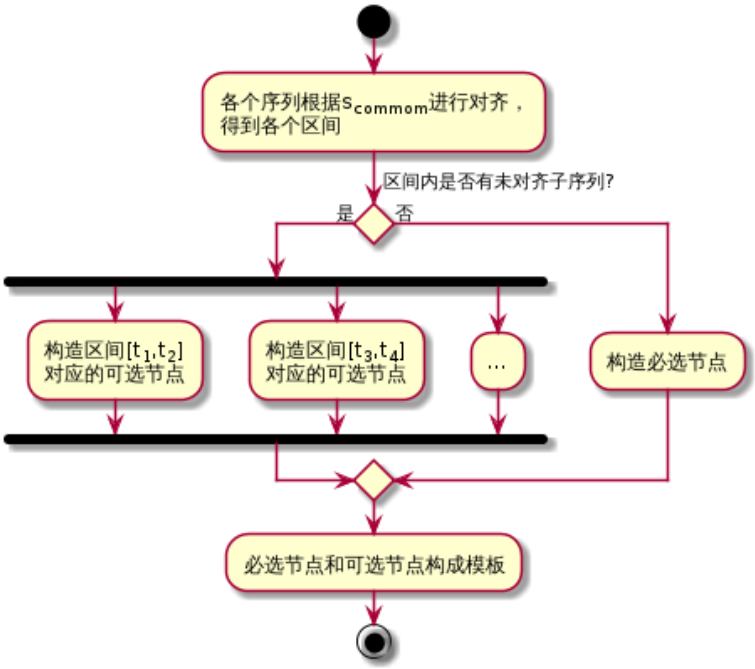


图 5.2　模板生成子模块整体流程图

thu。对于每一个区间 $[t_1, t_2]$，如果文档集合中的某些序列有未对齐的部分，即 $\exists s_i, |s_i[t_1, t_2]| > 0$，我们将对应生成一个可选节点 ON_{t_1,t_2}，这个可选节点对应于这个区间中所有未对齐的子序列。在我们的例子中，我们生成以下三个可选节点：

$$\begin{aligned} ON_{a,b} &= thu, 2/3 \mid orz, 1/3 \\ ON_{b,c} &= eat, 2/3 \\ ON_{d,e} &= lx, 2/3 \mid pku, 1/3 \end{aligned}$$

对齐的部分则对应生成必选节点，分别为：a, b, cd, e。将这些必选节点和可选节点间隔地依次组成一个序列，这 3 个序列对应的模板就生成好了。

由以上的例子，我们可以得到模板生成子模块的整体流程图如图 5.2所示，

然而，实际的情况却要比上面的例子更复杂一些。对于实际的网页文档集合，区间 $[t_1, t_2]$ 中未对齐的那些标签子序列，有些可能差异很大，有些则可能非常相近，但不完全一样。因此，我们需要将图 5.2中构造可选节点的过程进一步细化：先将这些子序列中差异较大的几种模式分开，然后针对每个分好的类别提取其中共有的部分，用于构造可选节点。

根据以上的叙述，我们发现，构造可选节点的这个子问题和我们的系统要

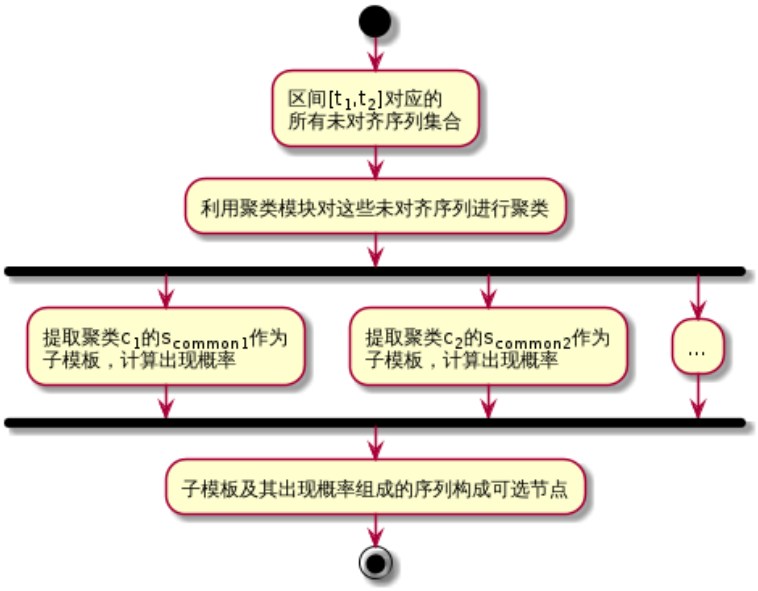


图 5.3　细化的构造可选节点的流程图

解决的问题有很大的相似性，都可以简单描述为：存在一个序列集合，里面的元素由几种不同的模式生成，需要先将这些元素分成几种类别，然后针对每个类别去提取公共的部分，即“模板”。对于构造可选节点这个子问题而言，这里的“模板”指的是在那些子序列的聚类中反复出现的模式，和我们要最终要提取的网页模板有很大的相似性。为了叙述简便，我们不妨称之为“子模板”。

提取这些子模板的过程和系统的整体框架极其类似。对于每个存在未对齐子序列的区间 $[t_1, t_2]$：

1. 区间内所有未对齐的序列组成一个新的序列集合 $set_{s'}$
2. 把集合 $set_{s'}$ 放入之前的聚类模块中，根据算法 3得到聚类的结果 *clusters*
3. 针对 *clusters* 中的每个聚类 c，使用算法 4得到它的“子模板”，并根据聚类 c 的大小和原始文档集合之间的大小，计算这个“子模板”的出现概率 p

这样，区间 $[t_1, t_2]$ 得到一个由多个子模板和其对应的出现概率组成的序列，这个序列就组成了区间 $[t_1, t_2]$ 所对应的的可选节点。必选节点则根据对齐结果生成，主要是将一些连续的中间没有未对齐序列的模板节点合并成一个必选节点，如上面的例子中，我们将 c, d 两个模板节点合并成了一个必选节点。这些必选节点和可选节点组成的序列也就构成了我们要处理的这个聚类的模板了。

细化以后的构造可选节点的流程如图 5.3所示。

最后，观察到从图 5.2中构造各个可选节点的过程是并行的，因此为了加快模板的计算速度，我们在这里再次使用了 Actor 模型并行地计算每个可选节点。

我们对每一个聚类都生成一个模板，这些模板将作为我们系统训练的输出，用于提取新的网页中的内容。

5.3 内容提取

这个子模块中，我们将利用聚类模块得到的聚类和前一个子模块生成好的模板，对新输入的网页进行内容的提取。

作为内容提取的第一步，我们需要先针对每个模板做一些简单的人工标注。由于整个模板生成的过程是无监督的，中间也没有考虑模板对应部分的语义信息，因此，在自动生成好模板后，还需要通过人工指定语义的办法使得系统能够判断哪部分的内容才是我们真正所关心的。比如当我们生成好了一类新闻网页的模板后，我们可以手工在模板的对应部分标上“标题”，“正文”和“评论”等，这样后续的内容提取部分就能自动将这些内容提取出来。

第二步需要先将新输入的网页经过预处理模块进行处理，包括去掉无用标签、合并重复记录等等，得到一个新的输入序列，计算这个序列和我们得到的每个聚类中心的距离，得到这些距离的最小值。如果这个最小值大于聚类时设置的阈值，则认为出现了一个新模板生成的网页，将其暂时存储，不进行处理。否则，我们选择与之距离最近的聚类对应的生成好的模板作为这个新的网页内容提取的模板。

第三步，模板生成模块类似，我们首先用模板中的必选节点对齐序列，然后对于每个有未对齐序列的区间，我们找到对应的可选节点，优先使用出现概率高的序列进行对齐。把所有的有未对齐序列的区间中的序列进行了对齐之后，那些原序列中尚未匹配的节点就是网页中动态生成的部分，也就是我们提取出来的内容。这时我们根据之前的人工标注，筛选出我们关心的部分，将其存为 XML 格式，作为输出。输入见

内容提取子模块的整体框架如图 5.4所示。

这里还需要补充两点：第一，由于之前合并了重复记录，因此在提取内容的时候需要进行复原，将其对应到原网页的多条内容上；第二，对于不属于任何类别的新的模板生成的网页，我们将其暂存，当其数量到达一定值的时候，

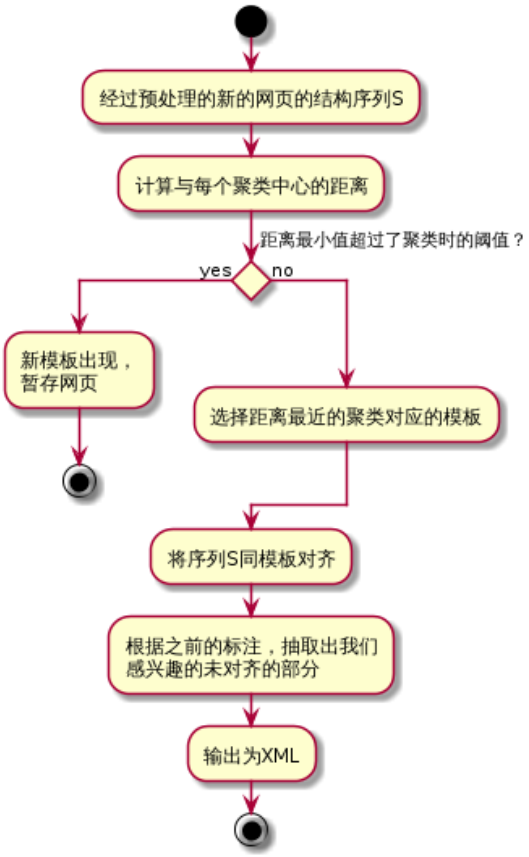


图 5.4　内容提取子模块的整体框架

我们将用这些网页进行训练，得到新的模板，加入到现有的系统中，从而实现了新模板的自动识别和生成功能。

5.4　本章总结

在这一章中，我们介绍了模板生成和内容提取模块的实现。为了讨论方便，我们首先形式化地定义了模板，然后我们介绍了如何对每个聚类生成对应的模板，重点在于如何正确构造模板中的可选节点。最后我们介绍了内容提取模块的具体实现流程。

至此，我们系统的各个模块的具体实现都已经介绍完了。

第 6 章　系统实现和实验结果

6.1　系统具体实现

我们采用 Scala 语言来搭建我们的系统。Scala 是一种运行在 JVM 上的静态语言，支持函数式、面向对象等多种编程范式，可以和 Java 代码无缝地进行交互。目前 Scala 还处于不断发展中，新版本不向后兼容，我们的系统在 Scala 2.10 上可以编译通过。在工程管理上，我们使用 sbt 管理 Scala 代码的编译和依赖关系。

在系统的实现中我们还使用了许多第三方库，主要有：

- 日志系统：基于 twitter 公司开源的 util 包，地址在 https://github.com/twitter/util。
- 网页字符集检测：ICU4J，地址在http://site.icu-project.org，这是一套 UNICODE 相关的工具集，提供了较好的字符集编码检测功能，我们用于检测中文网页的字符集。
- HTML 解析器：jsoup，地址在http://jsoup.org，是一个用 Java 语言实现的 HTML 解析器，用于 DOM Tree 的构建和节点内容的抽取。
- Actor 库：Akka。在第 4章中已经做了介绍。

6.2　实验演示系统

为了直观地显示网页模板的匹配效果，我们基于 Play! Framework 的 Web 开发框架实现了一个 Web Service，用于实验的演示。该 Web Service 的工作原理如图 6.1所示。

用户只需要输入一个需要和系统的模板进行匹配的网页的 URL，系统将返回一个修改过样式的新的网页，使得原网页和模板的匹配情况可以从浏览器中直观地看到。

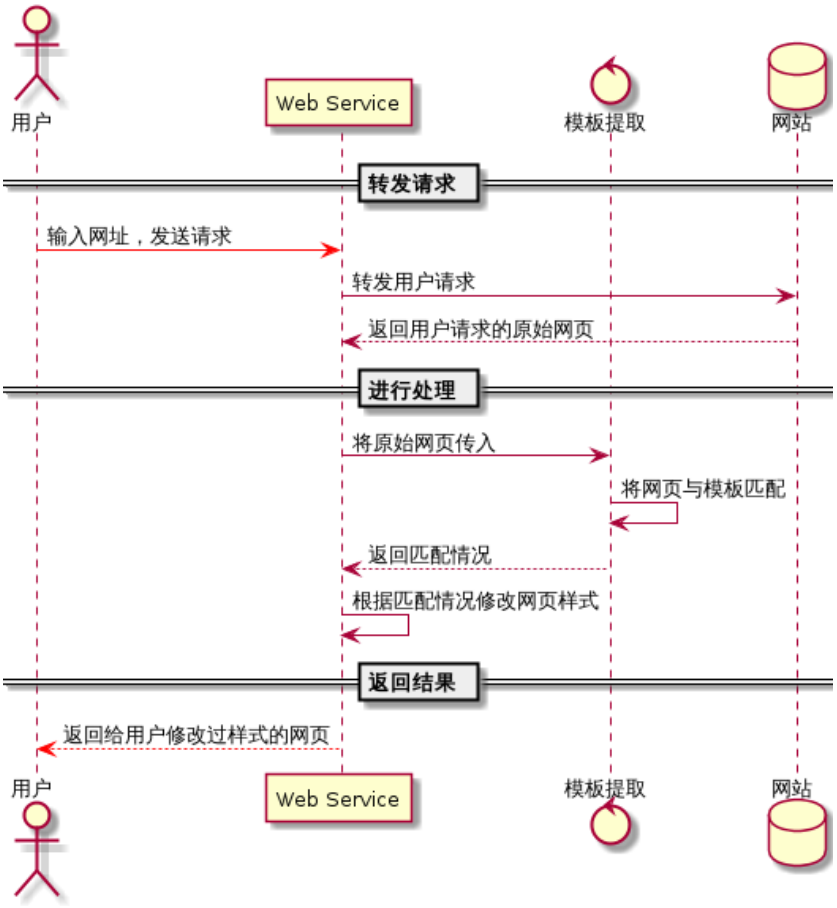


图 6.1　Web Service 工作原理

	blog	news	other
文件个数	59998	81561	183635
总大小	5.4G	7.9G	18G
来源	blog.sina.com.cn	news.xxx.com	

表 6.1　实验数据概况

6.3　实验数据和实验环境

实验数据的概况如表 6.1所示。

以下是我们实验中一些主要参数的的设置情况：

6.4　实验结果

我们将分模块对每个

聚类相似度阈值　0.7
训练的样本数
用于抽取的数据
表 6.2　主要参数设置

6.5　结果分析和存在的问题

第 7 章　工作总结和未来展望

7.1　工作总结

在这篇论文中，我们实现了一个大数据环境下网页模板自动聚类和提取系统。整个系统主要由三个模块组成：预处理，网页结构相似度计算和模板生成和内容提取，同时这三个模块也是我们整个系统的重点和难点。此外，还实现了一个用于实验演示的 Web Service。

本文的工作可以总结成以下几点：

1. 设计并实现了一个完整的系统。针对大数据的特点，在每个模块具体实现的时候充分考虑了效率问题，做了很多优化；实现了较高程度的自动化处理，最大限度减小人工参与的工作量。
2. 高效地实现了后缀树这个数据结构，在此基础上设计了一套适用于在树的先序遍历序列中查找重复子树的算法。
3. 实现并改进了最长公共子序列算法，将其用于计算文档的结构相似度；实现了简单的凝聚层次聚类算法，并将其用于文档的聚类
4. 实现了一个无监督的模板生成算法，通过人工指定模板中某些部分对应的语义，可以使用模板提取新的由该模板生成的网页中对应的信息。
5. 实现了一个 Web Service，可以直观地看到网页和模板的匹配情况。

7.2　未来工作展望

未来的工作可以针对系统已经实现的主要的三个模块分别展开：

- 预处理模块：改进利用后缀树查找重复子树的算法，进一步提高算法运行效率。思路是改进 Ukkonen 原有的在线构造算法，在构造树的时候就考虑一些原本的结构信息，尽早过滤掉一些不可能的序列。目前后缀树中还有一些信息也可以加入到对重复序列的判断上，比如后缀链。
- 网页结构相似度计算和网页聚类模块：可以进一步改进算法，包括计算相似度的算法和聚类算法。在计算最长公共子序列的时候还可以加入一些其

他的除了深度之外的更多结构信息；可以考虑使用更复杂但是鲁棒性更好的一些聚类算法。

- 模板生成和内容提取：对模板生成算法做一些修改，比如可以考虑采用半监督的学习方法，人工标注一些数据，减小数据中噪音的影响，增加算法鲁棒性；也可以在模板生成过程中设计一些更高级的机器学习算法。在内容提取的时候，可以采用树的匹配算法而不是序列的匹配算法。

插图索引

表格索引

参考文献

[1] 新华网. 新浪微博注册用户突破 3 亿每日发博量超过 1 亿条

[2] 新华每日电讯. 新浪微博用户注册数超 5 亿

[3] Kalakota R. Big Data Infographic and Gartner 2012 Top 10 Strategic Tech Trends, 2011. http://practicalanalytics.wordpress.com/2011/11/11/big-data-infographic-and-gartner-2012-top-10-strategic-tech-trends/

[4] Laney D. 3D Data Management: Controlling Data Volume, Velocity and Variety. Technical report, Gartner, February 06, 2001. http://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf

[5] Tai K C. The tree-to-tree correction problem. Journal of the ACM (JACM), 1979, 26(3):422–433

[6] Reis D D C, Golgher P B, Silva A, et al. Automatic web news extraction using tree edit distance. Proceedings of Proceedings of the 13th international conference on World Wide Web. ACM, 2004. 502–511

[7] Zhang K, Shasha D. Simple fast algorithms for the editing distance between trees and related problems. SIAM journal on computing, 1989, 18(6):1245–1262

[8] Dubiner M, Galil Z, Magen E. Faster tree pattern matching. Journal of the ACM (JACM), 1994, 41(2):205–213

[9] Touzet H. A linear tree edit distance algorithm for similar ordered trees. Proceedings of Combinatorial Pattern Matching. Springer, 2005. 334–345

[10] Gottron T. Clustering template based web documents. Advances in Information Retrieval, 2008. 40–51

[11] Joshi S, Agrawal N, Krishnapuram R, et al. A bag of paths model for measuring structural similarity in Web documents. Proceedings of Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2003. 577–582

[12] Buttler D. A short survey of document structure similarity algorithms. United States. Department of Energy, 2004

[13] Kim C, Shim K. Text: Automatic template extraction from heterogeneous web pages. Knowledge and Data Engineering, IEEE Transactions on, 2011, 23(4):612–626

[14] Flesca S, Manco G, Masciari E, et al. Detecting structural similarities between XML documents. Proceedings of Proc. of the Inter. ACM SIGMOD Workshop on The Web and Databases (WebDB), 2002. 55–60

[15] Arasu A, Garcia-Molina H. Extracting structured data from web pages. Proceedings of Proceedings of the 2003 ACM SIGMOD international conference on Management of data. ACM, 2003. 337–348

[16] Carlson A, Schafer C. Bootstrapping information extraction from semi-structured web pages. Proceedings of Machine Learning and Knowledge Discovery in Databases. Springer, 2008: 195–210

[17] Cai D, Yu S, Wen J R, et al. VIPS: a visionbased page segmentation algorithm. Technical report, Microsoft technical report, MSR-TR-2003-79, 2003

[18] Weiner P. Linear pattern matching algorithms. Proceedings of Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on. IEEE, 1973. 1–11

[19] Ukkonen E. On-line construction of suffix trees. Algorithmica, 1995, 14(3):249–260

[20] Hewitt C, Bishop P, Steiger R. A universal modular actor formalism for artificial intelligence. Proceedings of Proceedings of the 3rd international joint conference on Artificial intelligence. Morgan Kaufmann Publishers Inc., 1973. 235–245

致　谢

衷心感谢导师 xxx 教授和物理系 xxx 副教授对本人的精心指导。他们的言传身教将使我终生受益。

在美国麻省理工学院化学系进行九个月的合作研究期间，承蒙 xxx 教授热心指导与帮助，不胜感激。感谢 xx 实验室主任 xx 教授，以及实验室全体老师和同学们的热情帮助和支持！本课题承蒙国家自然科学基金资助，特此致谢。

感谢 ThuThesis，它的存在让我的论文写作轻松自在了许多，让我的论文格式规整漂亮了许多。

声　明

本人郑重声明：所呈交的学位论文，是本人在导师指导下，独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。

签　名：_____ 日　期：_____

附录 A　书面翻译

文档结构相似性算法调研

摘要

这篇论文对文档结构相似性算法做了简明的调研，包括优化的树编辑距离算法和各种近似算法。这些近似算法包括简单加权标签相似度算法，文档结构的傅里叶变换，和将连续序列技术应用到结构相似度计算上。我们展示了三个令人惊奇的结果。第一，傅里叶变换的方法是所有近似算法中最不精确的一个，同时也是最慢的一个。第二，优化的树编辑距离的算法并不一定是最好的用来将不同网站的网页进行聚类的算法。第三，对于许多应用来说，最简单的结构的近似可能是最有用也是最有效率的机制。

A.1　简介

随着万维网上大量的文档的出现，自动地处理这些文档，将其应用于信息抽取，近似聚类和搜索的需求越来越大。在这个领域的主要工作主要集中在文档的内容上。然而，虽然万维网的继续发展和进化，越来越多的信息被放在结构化的富文本中，从 HTML 转换到 XML。这个结构化的信息是一个文档意义的重要体现。从文档中辨别出结构上“相似”，或者结构上互相“包含”的那些文档的方法是一个非常重要那些相关的相似文档关联起来的机制，而这些文档可能包含不同的文本内容，那些基于文本内容的相似度算法起不到这样的作用。

现在已有几个文档结构在其中起到关键作用的应用。目前的信息提取算法隐式或显式地依赖文档的结构化元素。结构化的信息能帮助我们将大量的从不同网站上获得的网页整理成一些可以大致可以比较的集合。这就使得软件能够将可以抽取出正确结果的集合和那些不能抽取有用信息的集合分离开来。

结构相似度是一个非常重要的话题，有非常多的算法可以计算任意两个文档结构之间的最小编辑距离。然而，由于这些算法复杂度非常高，通常都需要

$O(n^2)$ 或者更多的时间去计算距离，因此创造一些更快的，但是距离的计算精确度有些许损失的算法是有可能的。

我们在这篇论文的第二节展示了当前用于检测结构相似度的算法的概述。之后在第三节我们描述了一个新的基于 shingle 的计算结构相似度的近似算法。在第四节我们在速度和精确度上对比了一下这些近似度算法和优化的树编辑距离算法。在第五节中我们用对不同算法特点的概括进行了总结。

A.2 当前研究状况

基于树的文档结构性相似度的研究已经有很长的历史了。早期基于树的变换检测来自 [1]，[2]。近期 Shasha [3]，[4]，[5] 和 Chawathe [6]，[7] 做了一些关于树到树变换的研究，主要集中在如何创建一个最小的脚本用来进行树之间的转换。在将一些基于结构的相似度计算修改成半结构的格式方面也有很多的尝试，包括 NiagraCQ [8]，Xdiff [9]，适用于 XML 的 Xyleme [10]，[11]，以及 AIDE [12]，[13] 和适用于 HTML 的 ChangeDetector™。

之前关于 HTML 文档相似度的工作大部分集中在了内容相似度上，页面分段 [15] 技术也是一样。目前的结构相似度集中在了 XML 的 Schema 的相似度上。DTD 相似度研究集中在对成对的文档和未知但相似的 DTD 的近似度计算上。这要求每两篇文档比较一次需要 $O(n^2)$ 的计算复杂度。其他的工作将文档之间结构相似度的问题转化为用傅里叶变换的时间序列的相似度，实现上采用快速傅里叶变换以实现 $O(n \times lgn)$ 的复杂度。

在这篇论文中，我们引进了一个将 shingle 技术应用在衡量文档结构相似度上的方法。这要求用 $O(k)$ 的复杂度去创造一个 shingle 文档（在这里 k 表示节点的个数），以及提供常数时间的复杂度进行文档之间的比较。在计算时间上的节省是通过损失计算精确度的得到的，实际可以任意减小精确度以满足不同的要求。第四节比较了这个技术与其他近似算法在不同的数据集上的表现。

A.2.1 近似算法

这里我们提供一个对不同类型的用于计算文档相似度算法的概述。我们将描述的衡量标准包括树编辑距离，标签距离，傅里叶变换和路径相似度。shingle 技术的动机和算法我们将在第三节给出。

树编辑距离相似度。一些作者提供了一些计算两个树之间优化的树编辑距离的算法。这篇论文使用 Nierman 和 Jagadish [16] 描述的动态编程实现。一般来说，编辑距离衡量的是将一个树转换为另一个树所要求的插入，删除和更新的最少的节点个数。通过将编辑操作的次数和较大的那个文档的节点个数之间进行归一化，可以将其转换为相似度的衡量标准。令 N_i 为文档 D_i 的树形表示的节点集合，于是

$$TED(D_i, D_j) = \frac{editDistance(D_i, D_j)}{max(|N_i|, |N_j|)}$$

标签相似度。标签相似度可能是结构相似度最简单的度量方法，因为它只衡量两个标签集合之间的近似程度。在 XML 文档中，标签是 schema 的一个组成部分，使用一个类似标签集合的页面 schema 也很有可能类似。两个文档的标签集合可以用来测量它们的重合程度。令 T_i 为文档 D_i 所包含的标签集合，T_J 为文档 D_j 包含的标签集合。两个页面的简单的标签相似度为 T_i 和 T_j 的交集和并集的比例。

然而，由于各种原因，这并不令人满意。一个关键的问题是遵循一个相同 schema 的页面，比如 HTML，只有一个非常有限的不同标签的个数；一个页面可能包含非常多的一个特殊的标签，但是用于比较的页面只包含相对很少的这个标签。为了补偿标签频率，我们可以引入一个加权的相似度度量。令 t_{ik} 为 T_i 的一个成员，w_{ik} 为标签 t_{ik} 在文档 D_i 中出现的次数。此外，令 t_{jk} 为 T_j 中对应的标签，v_{jk} 为 t_{jk} 在文档 D_j 中出现的次数。如果有 n 个互不相同的标签同时出现在文档 D_i 和 D_j 中，加权标签相似度可以表示为：

$$WTS(D_i, D_j) = \frac{\sum_{k=1}^{n} 2 * min(w_{ik}, v_{jk})}{\sum_{k=1}^{n} (w_{ik} + v_{jk})}$$

由于这个仅包含每个文档中的标签集合，结构相似度的精确性与使用的标签无关。因此，这个度量方法在类似 HTML 的环境中应该只有非常低的精确度，这些环境标签集合是很有限的，但是结构却变化非常大。它在一些都遵循一个小的 schema 集合的文档上有可能变得更加精确，因为这些 schema 限制了文档结构的变化。

傅里叶变换相似度衡量。Flesca *et al.* 引入傅里叶变换作为计算文档相似度的一个手段。基本的想法是将一个文档的开始标签和结束标签以外的全部信息去掉，只保留一个表示结构的骨干。然后将这个结构转化为一个数字序列。将

这个数字序列视为时序序列，然后使用傅里叶变换将其转化到频域。最终，两个文档之间的距离可以通过两个信号的幅度差别进行计算。

这个算法有几个对结果有重大影响的关键组成部分。像一开始所说的，文档结构的编码采用一个独特的（序列的）正数来表示每一个开始标签，用负数来表示对应的结束标签。属性被当做标签。注意到这个意味着为了比较两个文档，标签的数字映射必须在每个文档中都是一样的。Flesca *et al.* 选择了一个文档 d 的多层的编码，将其映射成一个序列 $[S_0, S_1, \ldots, S_n]$，其中

$$S_i = \gamma(t_i) \times expF(t_i) + \sum_{t_j \in nest_d(t_i)} \gamma(t_j) \times expF(t_j)$$

其中 $\gamma(t_i)$ 是一个对应于第 i^{th} 个标签的整数，$expF(t_i) = B^{maxdepth(D)-l_{t_i}}$ 是一个决定标签权重的指数因子，B 是一个固定的底，$maxdepth(D)$ 是正在比较的文档的最大深度，l_{t_i} 是第 i^{th} 个标签的深度，$nest_d(t_i)$ 是 t_i 的祖先集合。

最终的两个文档 d_1, d_2 的距离度量通过傅里叶变换定义为

$$dist(d_1, d_2) = (\sum_{k=1}^{M/2} (|[FFT(h_1)](k)| - |[FFT(h_2)](k)|)^2)^{\frac{1}{2}}$$

其中 FFT 傅里叶变换关于同时出现在 h_1 和 h_2 中频率的插值，h_1 是 d_1 对应的信号，M 是出现在插值中的点的个数。

这个方法有一些困难的地方。第一，FFT 要求点的个数是 2 的幂。一个 DFT 实现使用了和文档的时序表示一样多的点，意味着 DFT 和 FFT 近似算法的精度是不一样的。对于我们的比较而言，我们发现了 FFT 当成 DFT 时是 $O(n^2)$ 的，在实践中它比树编辑距离算法更慢，而且 DFT 的精度比 FFT 的精度要低。

第二，调和标签映射的要求和预先计算用于比较文档的最大深度意味着需要有根据单一文档来预先计算这个算法的任意部分的能力，以减小成对比较所需的时间。

第三，傅里叶变换典型的用法是在重复的无限时间序列上。我们所碰到的那些文档是有限的。为了可以使用这个变换，我们必须将从文档中提取的时间序列扩展为无限重复的。对于原始文档这意味着什么以及这个对于比较有那些影响我们都不清楚。

A.3 路径 shingle

使用优化的树编辑距离算法的问题在于它在大的文档集上代价非常大。目前所展示的近似算法要么不够直观（傅里叶变换），要么只提供了一个粗略的相似度度量（加权标签算法）。我们感到需要有一个可以根据应用来调整精度的快速的近似算法。

Shingle 是 Broder 在 [15] 中为计算文本文件相似度和包含度而引入的技术。这个技术减小了文档中词或者标记的集合，将其变成一个散列的列表，能够直接和另一个文档进行比较，通过集合的差集，并集和交集来计算相似度或者包含度。

更进一步地，shingle 集合的子集，叫做 sketch，可以用于计算文档的相似度。直观地说，sketch 是一个页面的随机采样的一段文本。关键在于由于随机映射在所有的页面中都是一样的，而且结果是排好序的，这些样本是可以直接在不同的页面之间进行比较。页面采样的重合度意味着整个页面的重合程度。

我们将展示如何修改这个技术，将其应用到文档的结构中。这使我们可以在线性时间计算文档结构相似度。通过稍微减小精度，可以实现常数时间的任意大小的文档之间的比较。

A.3.1 路径相似度

为了创建一个适用 shingle 技术的结构编码，我们必须找出一个方法去创建一个标记的列表，用于表示这个结构。一些自然的选择，比如使用深度优先或者广度优先的堆编码，因为用于表示树的标记列表本身的不分段特性从而证明是不合适的。这意味着当一个分支结束和另一个分支开始的时候，覆盖序列的窗口不能被辨别出来。覆盖这些的窗口不能精确表示原始文档的任一部分。为了找到用于这种编码的合适的分段方法，衍生了另一个自然的编码方法：路径。

半结构化的文档（HTML 和 XML）可以被看成一个由分支和从根到叶的路径组成的序列。为了我们的目标，我们将任意的部分路径，即从根到文档中任意的节点组成的路径也认为是一个标记。一个树可以用一个这些标记的序列来表示。比如，HTML 中最简单的树有一个 title 和 body 元素，可以被编码为：

/html

/html/head

/html/head/title/

/html/head/title/[text]
/html/body /html/body/[text]

路径相似度衡量两个不同文档之间的路径的相似度。一个路径被定义为一个从根节点开始，到叶子节点结束的相连的节点列表。路径相似度可以从多种方式进行度量：二元的，路径要么相等或者不相等；部分的，在每个路径中可以比较的节点的数目是知道的；或者加权的，节点根据它到根的距离进行加权。

部分路径相似度需要很大的计算代价。因为两个树的路径有 $n!$ 种可能的映射，用穷举的算法来产生优化的相似度分数是不可行的。二元的相似度代价要小很多，因为在一个版本中的每一个独特的路径可以通过数据库的 join 技术来匹配另一个版本中的等价路径。相似度可以通过匹配上的路径个数和两个树总的路径数的比例来计算。

A.3.2 将 shingle 应用到路径上

一个 shingle 是一个从文档中得到的连续的标记的子序列。两个文档的相似性定义为

$$r(D_i, D_j) = \frac{S(D_i, w) \cap S(D_j, w)}{S(D_i, w) \cup S(D_j, w)}$$

然而 $S(D_i, w)$ 是从文档 D_i 创建长度为 w 的 shingle 的操作符。类似地，文档 D_i 和 D_j 的包含度定义为

$$c(D_i, D_j) = \frac{S(D_i, w) \cap S(D_j, w)}{S(D_i, w)}$$

为了方便和快速的处理，shingle 可能通过散列函数转换成数字。这个散列函数应该有较高的可信度使得两个 shingle 散列成一个数的冲突的可能性变得很小。让散列的空间显著大于 shingle 的空间可以使得构造这样一个合适的散列函数变得容易很多。依据一个 shingle 中标记的个数（或者窗口的长度），这可能会非常微妙。

一个将结构化比较的复杂度减小到 $O(1)$ 的关键技术是每个文档只保留一个相对小的 sketch。从 [15] 中我们可以看到从一个 shingle 集合的排列中进行随机采样得到的样本可以被当成两个文档之间相似度的无偏估算子。一个有效的实现这个的办法是通过伪随机数生成算法来散列值，将结果排序，然后选择最小的 k 的结果。

为了将 shingle 应用到路径结构中，我们定义 $S(D_i, w)$ 如下：对于 D_i 的树形表示中的每个节点，计算从根到该节点的路径；根据这个（部分）路径的标签名列表创建一个散列；将该散列加入到当前窗口；窗口向前滑动一个单位（即将窗口的第一个散列值去掉）。。

注意到根据定义，shingle 集合可能是一个集合或是一个包——类似于标签相似度和加权标签相似度之间的差别。于是就出现了使用一个集合来包含 shingle 会显著减小 shingle 的表达能力，并将更大的误差引入到估计中。

我们测量了路径 shingle 和部分路径 shingle 的精度。比较是在使用无穷的 k 的不同的窗口大小下进行的。用来比较是数据是从 `my.yahoo.com` 下载的在两年时间内的典型的网页快照。结果显示小的窗口大小（从 1 到 4）对精度没有影响：两个聚类都没有错误，四个聚类只引入百分之三的错误。用来比较的 shingle 个数的不同并不影响聚类的效果。k 的值从 10 测试到了 1000, 以及一个无穷数量的 shingle。所有的都显示了错误在百分之十之内。

A.4 实验

在这一节中我们经验性地评价不同的近似算法与树编辑距离相比的精度，同时还比较了不同的近似算法的性能。

所有的实验都是在一个配置了速龙双核 2G 赫兹的处理器，2.4Linux 内核的工作站上运行。算法使用 Java 实现的，在 Sun 1.4.2 JVM 上执行。所有的算法都采用了页摘要数据结构来实现，该数据结果比 DOM 树表示有很大的性能提升。性能的测量是取 10 次运行的平均值。

根据聚类进行比较。聚类是在树编辑距离的基础上用来衡量不同度量之间的效果的。树编辑距离是可证明的一个两个树之间的最佳的编辑距离。我们假定这是最好的相似度度量。其他的算法会产生一些不同的距离度量，不能直接和一个编辑距离进行比较。然而，如果一个大的文档集合是根据某个度量方法来聚类的，这些通过不同的度量方法产生的聚类是可以通过给定同一个聚类方法进行比较的。换句话说，如果两个文档通过树编辑距离判断是相似的，那么其他的度量也应该认为这两个文档是相似的，或者恰恰相反。

我们可以将一个在一个近似算法中被放在一个 shingle 聚类中，而在树编辑距离算法中放到另一个类中的归类成错误文档。这个错误度量有一些缺点。比

如，一个文档集合被分成两个聚类，任何度量都会有严格小于 50% 的最大错误率。一般地，随着聚类数目 n 的增长，最大错误率严格小于 $1 - \frac{1}{n}$。

我们用两个数据集进行聚类。第一个是从 500 个 XML 文档中综合生成的集合。这个集合建模一个书仓库，每个文档都列出一个书对应的作者，发行商和发行时期。文档之间唯一的结构差别在于这本书的作者的个数。

每个度量方法都用来度量两个文本之间距离。文档根据这些距离通过开 *k – means* 进行聚类。通过每种度量得到的聚类与树编辑距离得到的聚类进行比较，计算一个错误估计。结果显示在 Table I 中。

TABLE I

CLUSTER ERROR RATE OVER BOOK DATA

Similarity Metric	Error Rate 6 clusters	4 clusters	2 clusters
Weighted Tag	6%	5%	2%
FFT	60%	46%	47%
Path	0 %	0%	2%
Path Shingles	6%	5%	2%

我们测试了最多 6 个聚类，因为这个数据集中根据书籍的作者数量，有 6 个自然的聚类。我们期望加权标签度量有较低的错误率，因为这些数据之间的结构差异只有 `author` 这个标签出现的次数。

第二个数据集是一系列从以下网站得到的快照：`cnn.com`，`corona.bc.ca`，`news.gnome.org`，`10-10phonerates.com` 和 `my.yahoo.com`。快照是 2001 到 2003 两年期间的，大概是每天一次。冗余的快照（由 MD5 签名决定）被移除了，每个站点的快照集合取样 20 份页面。一样的聚类算法用在这些文档上，只有这次我们有一个预定义好的聚类（通过网站），并且可以将每个算法同这些预定义好的聚类进行比较。结果显示在 Table II 中。

TABLE II

CLUSTER ERROR RATE OVER WEB DATA

Similarity Metric	Error Rate 6 clusters
Weighted Tag	0%
Path	28%
Path Shingles	34%
TED	38%
FFT	45%

这个错误率好像异常地高，特别是对于我们在其他测试中用来当做基准的树编辑距离算法。我们推测这个错误率是由于 HTML 的结构中用来呈现给用户内容的词汇相对较小。路径和路径 shingle 度量比树编辑距离性能要好，但也比预期的要差。这可能是因为它们使用了部分路径来描述树的结构。很深的树在顶部区域可能会呈现很多相似的结构，导致两个来自不同站点的树的相似度可能会互相偏向。FFT 度量方法的糟糕的性能不能给出一个简单的解释。可以说依据相同的最开始的一批标签（`html`，`head` 和 `body`），每个网页都是极度相似的信号，在叶子层面也是相似的构造（标签列表和文本）。然而，这个转换使得我们难以找出是网页的哪一些特征导致它们变得如此相似。

作为最后的比较，我们检查了以上表格中从一个站点来的快照。这会在比如监测一个页面随着时间变化的时候很有用。我们选择 `my.yahoo.com` 因为里面的内容会定期发生变化，但是结构上随着时间的变化很缓慢（比如，当一个新的图片被加入到一些特点的假日）。我们再次把使用近似算法和树编辑距离产生的聚类进行比较。

TABLE III

CLUSTER ERROR RATE OVER YAHOO! DATA

Similarity Metric	Error Rate 6 clusters	4 clusters	2 clusters
FFT	33%	31%	29%
Path	59%	46%	1%
Weighted Tag	60%	50%	1%
Path Shingles	61%	47%	1%

我们观察到除了 FFT 以外的大部分近似算法在小的聚类大小上都有非常低的错误率，但是错误率在聚类的规模变大时会激增。观察一下描述树编辑距离聚类和 FFT 聚类以及路径/路径 shingle 聚类之间映射变化的矩阵是有意义的。Table IV 描述了 FFT 聚类和树编辑距离聚类的不同，Table V 描述了路径聚类和树编辑距离聚类之间的不同。

TABLE IV

CLUSTER COMPARISON BETWEEN FFT AND TED METRICS; 6 CLUSTERS

Cluster # FFT	TED 1	2	3	4	5	6
1	1	0	0	0	0	0
2	1	0	0	0	0	0
3	1	0	0	0	0	0
4	1	0	0	0	0	0
5	92	31	1	8	0	0
6	1	0	0	0	0	0

从这两个矩阵我们可以推导出 FFT 度量会倾向于将所有的数据点聚成一个类别，因此不提供一个有效的分辨相似网页的能力。在另一方面，路径度量

（路径 shingle 和加权标签度量实际是一样的）比树编辑距离更有分辨能力，能够比树编辑距离分出更好的类别。尽管这些实验中将这种情况标为了错误，它可能有很重要的分辨那些感觉上不一样，但在相同的编辑距离内的树的功能。

最惊奇的观察在于加权标签的相似度度量方法，一开始我们认为是一文不值的东西，却能和复杂很多的技术达到一样的精确程度。这可能可以归因为大部分实验是在来自一个站点的相对同构的页面上进行的（就像书籍数据集或者 Yahoo! 数据集的情况一样），或者可以归因为非同构的站点为了突出它们的对比而使用了不同 HTML 标签子集。另一个观察是傅里叶变换技术在很简单的数据集上表现也很差。这就使得我们认为尽管这是一个将结构转换为一个更简单的用于比较的格式的想法，但是它并不适用与文档结构，无论是从分析上还是实验上。

A.4.1 性能比较

选择一个近似算法的主要原因是为了将速度提高到一个可以接受的水平，因为最优的算法太慢了。文档聚类用于数据抽取或者搜索和获取方法，提供了一个结构化相似度的很好的例子。用于精度估计的相对小的数据集说明了近似算法在计算近似相似度上的有效性。Figure 1 展示了不同算法在书籍文档的数量对数增长时的相对代价。

聚类的时间是在一个小于一千比特的较小的文档上计算的。为了更好地理解计算两个文档之间差异的代价，我们比较了在 TCP-H 基准测试数据上的每个度量方法的代价。这个数据是由 Toxgene [19]XML 文档生成器随机生成的。我们修改了生成器的参数以便产生包含原始数据集合 1%，5%，10%，15%，20%，和 25% 的文档。这个生成器跑两次，在每个分数上产生两个不同的文档。结果在 Figure 2 中。

我们可以看到树编辑距离算法比任何其他的近似算法都要慢好几个数量级。FFT 算法也显示出了尽管它用很大的精度换来了速度，它仍然比加权标签或者路径 shingle 方法慢一个数量级，后两者的精度还显著更高。

大文件支持。以空间换时间的优化对于小到中型大小的文件都工作地很好，但是树编辑距离在其数据结构会超过物理内存的大文件上明显变慢。当物理内存被耗尽，机器被迫开始使用交换内存——这个会慢好几个数量级。

shingle 在创建常数大小的大文件指纹时有优势，消除了在计算时在内存中

维护复杂数据结构的需要。

shingle 还可以部分调优，就算是因此取了原始的指纹。给定一个窗口散列的集合，只有最前面的 k 个需要比较。用于比较的散列的数目可以调整，把精度用来换速度和空间。这就使得对于更低精度的比较，一次性可以有更多的指纹在内存中驻留。

A.5 总结

我们展示了一些测量文档结构相似度的算法，比较了它们的精度和性能。我们有了一些有趣的发现。

第一，我们提供了一个对 [17] 中所描述的傅里叶变换的实验性的批判。尽管相比最优化树编辑距离算法，傅里叶变换方法是一个更快的方法，但这个方法没有提供在不同情境中的一个精确的相似度衡量。此外，这个技术的性能也通常比其他更直观的相似度计算方法要差。

第二，对于很多结构相似度的应用来说，最简单的计算标签数量的方法提供了最好的性能情况下的一个可以接受的精度。我们一开始将加权标签相似度当成一个不重要的计算结构相似度最快的近似方法。然而，结果是它可以和其他任何近似算法表现一样好，甚至更好。尽管它没有像树编辑距离方法一样（匹配的相同子树），或者路径 shingle 方法一样（子结构包含性）提供一定的结构特征，但是对于不需要这些特征的应用来说，这个算法既快又有辨别力。

最后，我们基于文档结构中的路径展示了一个新的相似度度量。我们通过应用 shingle 技术，可以从任意的文档中创建常数大小的表示方法，使得比起其他相似度度量，聚类方法可以应用到大很多的文档集合中。此外，这种度量具有在一个大文档集合中搜索子结构和在基于树的文档集合中做一些结构化的挖掘的能力。

致谢

作者要感谢 Chunk Baldwin 和 Ghaleb Abdulla，他们刺激了这些话题之间的对话。作者还要感觉 Daniel Rocco，他搭建了第一个实验框架，并帮助开发了页面摘要数据结构，作为许多算法的基础。

这个工作是基于 No. W-7405- ENG-48. UCRL-CONF-202728 法令，在美国能源部的保护下，在 University of California Lawrence Livermore National Laboratory 进行的，

参考文献

[1] K. C. Tai, “The tree-to-tree correction problem,” Journal of the ACM, vol. 26, no. 3, 1979.

[2] S. Y. Lu, “A tree-to-tree distance and its application to cluster analysis,” IEEE Trans. Pattern Anal. Mach. Intell. PAMI-1, no. 2, 1979.

[3] D. Shasha and K. Zhang, “Fast algorithms for the unit cost editing distance between trees,” Journal of Algorithms, no. 11, 1990.

[4] K. Zhang, D. Shasha, and J. T.-L. Wang, “Approximate tree matching in the presence of variable length don’t cares,” J. Algorithms, vol. 16, no. 1, pp. 33–66, 1994. [Online]. Available: citeseer.nj.nec.com/zhang93approximate.html

[5] D. Shasha and K. Zhang, “Approximate tree pattern matching,” in Pattern Matching Algorithms. Oxford University Press, 1997, pp. 341–371. [Online]. Available: citeseer.nj.nec.com/95609.html

[6] S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, “Change detection in hierarchically structured information,” in Proceedings of ACM SIGMOD, 1996.

[7] S. S. Chawathe and H. Garcia-Molina, “Meaningful change detection in structured data,” in Proceedings of the 1997 ACM SIGMOD, 1997, pp. 26–37. [Online]. Available: citeseer.nj.nec.com/article/chawathe97meaningful.html

[8] J. Chen, D. DeWitt, F. Tian, and Y. Wang, “NiagaraCQ: A scalable continuous query system for Internet databases,” in Proceedings of the 2000 ACM SIGMOD, 2000.

[9] Y. Wang, D. DeWitt, and J.-Y. Cai, “X-Diff: An effective change detection algorithm for XML documents,” International Conference on Data Engineering, 2003.

[10] A. Marian, S. Abiteboul, G. Cobena, and L. Mignet, “Change- centric man-

agement of versions in an XML warehouse,” in The VLDB Journal, 2001, pp. 581–590. [Online]. Available: citeseer.nj.nec.com/marian00changecentric.html

[11] G. Cobena, S. Abiteboul, and A. Marian, “Detecting changes in XML documents,” in International Conference on Data Engineering, 2002, pp. 41 –52.

[12] F. Douglis, T. Ball, Y.-F. Chen, and E. Koutsofios, “The AT&T Internet difference engine: Tracking and viewing changes on the Web,” in World Wide Web, vol. 1, January 1998, pp. 27–44.

[13] Y.-F. Chen, F. Douglis, H. Huan, and K.-P. Vo, “TopBlend: An efficient implementation of HtmlDiff in Java,” in Proceedings of the WebNet2000 Conference, San Antonio, TX, November 2000.

[14] V. Boyapati, K. Chevrier, A. Finkel, N. Glance, T. Pierce, R. Stokton, and C. Whitmer, “ChangeDetector(TM): A site-level monitoring tool for the WWW,” in WWW2002, May 2002.

[15] A. Z. Broder, “On the Resemblance and Containment of Documents,” in Proceedings of Compression and Complexity of SEQUENCES 1997, 1997.

[16] A. Nierman and H. Jagadish, “Evaluating structural similarity in XML documents,” Fifth International Workshop on the Web and Databases, 2002.

[17] S. Flesca, G. Manco, E. Masciari, L. Pontieri, and A. Pugliese, “Detect- ing structural similarities between XML documents,” Fifth International Workshop on the Web and Databases, 2002.

[18] D. Rocco, D. Buttler, and L. Liu, “Page Digest for large-scale Web ser- vices,” in Proceedings of the IEEE Conference on Electronic Commerce, June 2003.

[19] D. Barbosa, A. O. Mendelzon, J. Keenleyside, and K. A. Lyons, “Toxgene: An extensible template-based data generator for XML,” in SIGMOD Conference, 2002. [Online]. Available: citeseer.nj.nec.com/525958.html