High-quality Lossless Web Page Template and Data Separation

Chenxu Zhao

Submitted in total fulfilment of the requirements of the degree of
Master of Philosophy

School of Computing and Information Systems
THE UNIVERSITY OF MELBOURNE

September 2018
Abstract

Web page separation is an important task that aims to separate a web page into template code and data records populated into the template. Web page separation needs to work in a lossless manner where the web page can be reconstructed by running the template code on the data records. In this thesis, we investigate two sub-problems of web page separation for obtaining (1) high-quality template code and (2) high-quality data records.

For the first sub-problem, we focus on improving the maintainability of the template code. Easily maintainable template code is reliable and will simplify further developments on top of the template code, e.g., to update the web templates. We formulate such a problem and analyze its complexity. We show that this problem is NP-hard. We then propose a heuristic algorithm to solve the problem. The main idea of our algorithm is to parse a web page into a tree and then to process it recursively in a bottom-up manner with three steps: splitting, folding, and alignment. In particular, we split siblings in the tree and fold them into chunks, where the alignment step is used to align sibling in the same chunk. During the sibling splitting step, to determine which siblings should be grouped into the same chunk, we further propose a population-based optimization algorithm named dual teaching and learning based optimization. We perform experiments on real data sets to evaluate the performance of our proposed algorithms in maximizing the maintainability of the template code produced. Experimental results show that our proposed algorithms outperform the baseline algorithms in the maintainability measure.

For the second sub-problem, we focus on extracting data records from a set of web pages which are generated by different unknown templates and deducing the schemas that
provide the data records. The extracted data records can be used in many applications, such as stock market prediction and personalized recommendation systems. We formulate such a problem and propose a framework to tackle the problem. Our framework processes web pages with four steps: web page template and data separation, template clustering, template alignment, and data record filtering. The web page template and data separation step separates web pages into template code and data records. The template clustering step then clusters the web pages by the similarity of template code. The template alignment step captures the differences among templates to construct a generalized template code which can generate all web pages in the same group. The data filtering step utilizes the template code to verify the data records extracted by the web page template and data separation step and modifies those which are incorrectly extracted. We perform experiments on real data sets to evaluate the performance of our framework. Experimental results show that our proposed framework outperforms baseline algorithms which assume a pre-known clustering of the set of web pages in the F-Score.
Declaration

This is to certify that

1. the thesis comprises only my original work towards the Master of Philosophy,

2. due acknowledgement has been made in the text to all other material used,

3. the thesis is less than 50,000 words in length, exclusive of tables, maps, bibliographies and appendices.

Chenxu Zhao, September 2018
Acknowledgements

First and foremost, I would like to express my profound gratitude to my principal supervisor Prof. Rui Zhang and my co-supervisor Dr. Jianzhong Qi for their knowledge, patience, guidance, and support during the pursuit of this Master’s degree. This thesis would not have been possible without them. I am deeply grateful to Prof. Ramamoohanarao Kotagiri and A/Prof. Michael Kirley who gave me guidance on combinatorial optimization algorithms and evolutionary algorithms. My progress would not have been that smooth without their constructive comments. I also wish to thank my advisory committee Chair, Dr. Atif Ahmad for watching my progress and giving me generous support during my study. Last but not least, I would like to thank all my colleagues, including Daochang Chen, Weihao Cheng, Yimeng Dai, Wei Gao, Wenkai Jiang, Yichen Li, Yuan Li, Zhen Wang, Yuan Xue, Jiayuan He, Aili Shen, Chuandong Yin, Yiqing Zhang, He Zhao and Yunxiang Zhao for all the pleasant memories.
Preface

One conference paper corresponding to the work reported in Chapter 3 of this thesis has been accepted:

- **Chenxu Zhao**, Rui Zhang and Jianzhong Qi, “Web Page Template and Data Separation for Better Maintainability,” accepted by International Conference on Web Information Systems Engineering (WISE), 2018. (WISE 2018 ranks A by the Computing Research and Education Association of Australasia, CORE)
To my parents, for their unconditional love.
# Contents

1 Introduction ............................................. 1
   1.1 Summary of Contributions .......................... 3
   1.2 Thesis Structure .................................. 4

2 Literature Review ................................. 7
   2.1 Web Information Extraction ....................... 7
   2.2 Tree Similarity .................................... 8
      2.2.1 DOM Tree ................................... 8
      2.2.2 Tree Similarity Measurements ............... 10
   2.3 Template Detection .................................. 13
      2.3.1 Regular Expression Based Techniques .......... 13
      2.3.2 XPath Based Techniques ....................... 16
      2.3.3 Tree Based Techniques ......................... 19
      2.3.4 Page-level Information Extraction ............ 24
   2.4 Software Maintenance .............................. 29
   2.5 Summary ........................................... 31

3 Web Page Separation for High-quality Template Code .......... 33
   3.1 Introduction ........................................ 33
   3.2 Preliminaries ...................................... 36
      3.2.1 SYNTHIA ..................................... 37
      3.2.2 Teaching and Learning Based Optimization Algorithm .................... 39
      3.2.3 Maintainability Index ........................ 40
   3.3 Problem Definition and Complexity Analysis .......... 41
      3.3.1 Problem Definition ............................ 41
      3.3.2 Complexity Analysis ........................... 42
   3.4 Our Method ........................................ 45
      3.4.1 Overall Algorithm Procedure .................. 45
      3.4.2 Splitting Algorithms .......................... 51
      3.4.3 Folding Algorithm ............................. 62
      3.4.4 Alignment Algorithm ......................... 63
   3.5 Experiments .......................................... 73
      3.5.1 Settings ........................................ 73
      3.5.2 Results ......................................... 75
      3.5.3 Effect of Parameters ........................... 79
List of Figures

2.1 An example of a DOM tree ............................................. 9
2.2 XPath pattern mapping ............................................... 17
2.3 An example of the partial tree alignment ......................... 21
2.4 Work Flow of FiVaTech ............................................... 26
2.5 An example of FiVaTech matrix alignment ....................... 28

3.1 An example of the web page separation problem ............... 34
3.2 An example of a DOM tree and its corresponding layout tree 38
3.3 Flowchart for TLBO .................................................. 39
3.4 An example of variation ............................................. 43
3.5 An example of the separation algorithm ......................... 48
3.6 A splitting example .................................................. 53
3.7 Workflow of dual-TLBO ............................................. 59
3.8 Running Example of dual-TLBO ................................... 60
3.9 Text nodes alignment .............................................. 65
3.10 Text node and reference node alignment ....................... 66
3.11 Element nodes alignment ......................................... 67
3.12 An example of null alignment of iteration nodes .......... 69
3.13 Effect of population size ........................................ 79
3.14 Effect of learning rate ............................................ 81
3.15 Effect of maximum generations ................................ 82
3.16 Effect of patience number ....................................... 83
3.17 Statistics of SYNTHIA-DATASET ............................... 85
3.18 Statistics of TBDW .................................................. 86
3.19 Statistics of UW-CAN .............................................. 88
3.20 Statistics of 4Universities ....................................... 89

4.1 An example of template components ................................ 94
4.2 Workflow of the general framework ............................... 97
4.3 A running example of overall algorithm ......................... 101
List of Tables

3.1 MI of Layout Trees ........................................ 41
3.2 Frequently Used Symbols ...................................... 50
3.3 Score .......................................................... 72
3.4 Alignment ..................................................... 72
3.5 SYNTHIA-DATASET ............................................ 76
3.6 TBDW .......................................................... 77
3.7 UW-CAN ....................................................... 77
3.8 4UNI-Faculty ................................................... 78
3.9 4UNI-Student ................................................... 78
4.1 Verticals of SWDE ............................................. 110
4.2 Experimental Results on TBDW ............................. 112
4.3 Experimental Results on SWDE-AUTO ...................... 113
4.4 Experimental Results on SWDE-BOOK ..................... 114
4.5 Experimental Results on SWDE-CAMERA ................. 115
4.6 Experimental Results on SWDE-JOB ....................... 115
4.7 Experimental Results on SWDE-MOVIE ................. 116
4.8 Experimental Results on SWDE-NBA ..................... 116
4.9 Experimental Results on SWDE-RESTAURANT .......... 117
4.10 Experimental Results on SWDE-UNIVERSITY ........... 117
4.11 Time Efficiency on SWDE ................................. 118
Chapter 1
Introduction

Web pages provide affluent information resources for a wide range of applications, such as web search engine [88] and personalized recommendation systems [63]. To access the information in web pages easily, many techniques have been proposed to extract structured information from web pages that are semi-structured or unstructured documents. For example, Carchiolo et al. [13], Adelfio et al. [1] and Alani et al. [2] investigate the hidden schema extraction task that aims to extract data records and their schema from web pages. Kundan et al. [54], Kayed et al. [49] and Zhai et al. [109] study the template detection task, which extracts the template that generated the web pages. Many other techniques are also related to web information extraction, such as wrapper maintenance [67], wrapper verification [16] and automatic web information extraction [82].

In this thesis, we study an important aspect of web information extraction named web page separation. Web page separation [73] is a task that aims to separate a web page into template code and data records populated into the template code. Web page separation needs to work in a lossless manner where the original web page can be reconstructed by running the template code on the data records. The quality of the template code is important because it affects further developments of the web pages, such as debugging and adding new content (features). Besides, high-quality data records with less noise save time for further processing, which is important for data-centric applications, such as recommendation systems [79] and stock market prediction [11]. The template code and data records are two important parts of the web page separation task. In this thesis, we study two sub-problems of the web page separation problem that aim to obtain (1)
high-quality template code and (2) high-quality data records.

For the first sub-problem, we study a web page separation problem that aims to extract easily maintainable template code from a web page. The problem is different from previous work \[73\] that aims to minimize the length of the template code and data records, i.e., to compress the web pages. Easily maintainable template code is important for reducing the software’s life cycle costs \[18\]. We formulate such a web page separation problem and analyze its complexity. We show that this problem is NP-hard. We then propose a heuristic algorithm to solve the problem. The main idea of our algorithm is to parse a web page into a tree and then to process it recursively in a bottom-up manner with three steps: splitting, folding, and alignment. In particular, we split siblings in the tree and fold them into chunks, where the alignment step is used to align sibling in the same chunk. During the sibling splitting step, to determine which siblings should be grouped into the same chunk, we further propose a population-based optimization algorithm named dual teaching and learning based optimization. We perform experiments on real data sets to evaluate the performance of our proposed algorithms in maximizing the maintainability of the template code produced. The experimental results show that our proposed algorithms outperform the baseline algorithms by 24.4% in the maintainability measure.

For the second sub-problem, we aim to extract data records from a set of web pages which are generated by different unknown templates. Our work is different from previous work \[49\], \[92\], \[93\] which assumes a pre-known clustering of the set of web pages. Extracting data records from web pages is an important task that provides rich data records for data-centric applications, such as stock market prediction \[107\] and personalized recommendation systems \[63\]. We formulate such a problem and propose a framework to tackle the problem. Our framework processes web pages with four steps: web page template and data separation, template clustering, template alignment, and data record filtering. The web page template and data separation step separates every web page into a template component and a data component. The template clustering step then clusters the web pages by the similarity of the extracted template components. For the template
1.1 Summary of Contributions

We summarize the contributions of this thesis.

The contributions on the problem of web page separation towards high-quality template code are summarized as follows:

- We propose a novel web page separation problem that aims to separate a web page into easily maintainable template code, and data records populated into the template code. We show that our web page separation problem is NP-hard.

- We propose a heuristic algorithm which processes a web page in a bottom-up manner with three steps: splitting, folding and alignment.

- We propose a population-based optimization problem, named dual teaching and learning based optimization (dual-TLBO), to find splitting that can maximize the maintainability of the template code after folding and aligning the web page source code.

- We propose a global alignment algorithm to capture the differences between different parts of the web page source code. It aims to maximize the maintainability of the template code after aligning the web page source code.
• We conduct an experimental study to evaluate the performance of our methods in improving the maintainability of web pages. Moreover, we evaluate our methods on web page compression. Our algorithms outperform the baseline algorithms by 24.4% in the maintainability measure.

The contributions on the problem of web page separation towards high-quality data records are summarized as follows:

• We formulate the problem that aims to extract data records from a set of web pages which are generated by different templates.

• We propose a framework that extracts data records from a set of web pages generated from different unknown templates.

• We propose a similarity measurement algorithm that utilizes both the semantics of data records and the structure of template code to measure the similarity among templates.

• We propose a template clustering algorithm that can recognize web pages which are generated by the same template.

• We conduct an experimental study to evaluate the performance of our framework in extracting data records. Moreover, we evaluate the recognition of entities within the data records. Our framework outperforms baseline algorithms which assume a pre-known clustering of the set of web pages.

1.2 Thesis Structure

The rest of this thesis is organized as follows:

• Chapter 2 reviews related work. We focus on closely related studies on template detection and data record extraction.

• Chapter 3 studies a web page separation problem that aims to improve the maintainability of the template code. We first formulate our problem and introduce a
new concept, variation, to analyze the complexity. Then, we propose a heuristic algorithm to solve our separation problem. Besides, we propose a population-based optimization algorithm, named dual teaching and learning optimization algorithm to perform the splitting procedure. Furthermore, we propose a global alignment algorithm to maximize the maintainability index of template code after aligning the web page source code. Finally, we perform an experimental study to evaluate our method on improving the maintainability of template code and web page compression.

- Chapter 4 studies a web page separation problem that aims to extract data records from a set of web pages which are generated by different unknown templates. We formulate our problem and propose a framework to tackle the problem. The framework consists of four steps: web page template and data separation, template clustering, template alignment and data record filtering. To align the templates and data records populated into, we further propose an algorithm utilizing the semantics of data records and the structure information of templates. Finally, we perform an experimental study to evaluate our method on data record extraction and deducing the schema of data records.

- Chapter 5 summarizes this thesis. We emphasize our findings and outcomes. Besides, we discuss future work and possible development of web page separation.
Chapter 2

Literature Review

In this chapter, we review studies which are close to the web page separation problem. In Section 2.1, we review the development of web information extraction and important studies in this area. In Section 2.2, we then review an important document model named DOM tree to represent a web page, and tree similarity measurements which are used to align sub-DOM trees. Section 2.3 reviews techniques on detecting the templates of web pages. We also briefly discuss conventional software maintenance techniques in Section 2.4. Section 2.5 summarizes this chapter.

2.1 Web Information Extraction

The Internet provides abundant data for various studies and applications, such as natural language processing [5] and stock prediction [107]. The data records in web pages cannot be directly used because most of them are unstructured and semi-structured. Web Information Extraction [6, 15, 17] is an important task that aims to format unstructured data and semi-structured data stored in web pages into structured data. To extract data records from web pages efficiently, wrappers [55, 67] are introduced. A wrapper is a program that can automatically extract data records from web pages. Wrappers usually utilize the HTML [24] syntax and the format of data records as features. Okada et al. [71] give a method that converts the web page into a tree using HTML syntax and constructs the wrapper using the paths from the root to data records. Li et al. [62] propose a method that construct regular expressions [97] as wrappers based on the format of data records.
As wrappers usually depend on the HTML syntax and the format of data records, the performance of wrappers drops significantly once the web pages are changed. To tackle this problem, Kim et al. [51] and Kayed et al. [49] give methods that automatically recognize the data records in web pages and extract data records from web pages. Generally, compared with the data wrappers which are designed for specific web pages, the automatic data extractor cannot achieve the same performance [34].

2.2 Tree Similarity

In Section 2.2.1, we first briefly review the DOM tree model. We then describe tree similarity measurements in Section 2.2.2.

2.2.1 DOM Tree

Web pages are written in HTML [24], which is an XML-like language. For example, Listing 2.1 is a piece of web page source code written in HTML. The HTML tags, such as “⟨div⟩” and “⟨li⟩” in Listing 2.1, wrap the data records (“School of Computing and Information Systems”, cf. Listing 2.1). HTML tags may contain attributes. For example, in Listing 2.1, “class” and “href” are the attributes of the “li” tag. Processing web pages written in HTML as string is inefficient because of the flexibility of the syntax of HTML. To tackle this problem, W3C proposes the Document Object Model (DOM) [100].

Document Object Model is an interface that organizes a web page as a tree structure where each tree node forms part of the web page. With the help of the Document Object Model, we can parse the source code of a web page into a DOM tree. The DOM tree is a rooted and ordered tree which consists of element nodes and text nodes. An element node contains a tag string and a list of attributes. Each attribute is a mapping from an attribute name to an attribute value, which is a string. A text node is a node with a textual value. To help understand the DOM tree, we give a running example to show how to parse a piece of web page source code (cf. Listing 2.1) into a DOM tree (cf. Figure 2.1). Due to the space limit, we hide the attributes of element nodes. In the DOM tree, “⟨div⟩”
and "⟨li⟩" are two element nodes and their tag string are "div" and "li" respectively. "⟨div⟩" is the parent of "⟨li⟩". The node "⟨li⟩" contains two attributes "class" and "href" whose values are "unimelb" and "https://cis.unimelb.edu.au/" respectively. “School of Computing and Information Systems” is a text node. We thus parse the source code of the web page into a DOM tree.

Listing 2.1: Web Source Code

```html
<div>
  <li class='unimelb' href='https://cis.unimelb.edu.au/'>
    School of Computing and Information Systems
  </li>
</div>
```

![Figure 2.1: An example of a DOM tree](image)

In our work, we need to identify the components of a web page that contain similar content. The components containing similar content are usually generated by the same template. These components are likely to be generated by running the template code on the data records from the same table of a database. The task of identification of web components containing similar content is named template detection. Most of the existing template detection techniques [49, 54, 109] utilizes tree structure similarity to find sub-DOM trees that have a similar structure. Intuitively, sub-DOM trees generated by the same template usually have similar structures. We describe representative tree similarity
measurements in the next section (Section 2.2.2).

One issue of the tree based template detection is that the web components that have similar structures may not be generated by the same template. The content within web components with similarity structure may be different. We address this problem and propose a similarity measurement that utilizes both the structure of web components and the semantics of data records within the web components. We detail the algorithm in Section 4.4.2.

2.2.2 Tree Similarity Measurements

We begin with a tree similarity measurement named *tree edit distance*. Tree edit distance \[9\] is the minimum-cost sequence of node edit operations that transform one tree into another tree. In particular, node edit operations are shown as follows:

- **insert**: insert a node as a child of an existing node and be the parent of a part of consecutive children of the existing node.

- **delete**: delete an existing node and add its children to its parent while maintaining the order of its parent’s children.

- **rename**: rename the label of an existing node. The label of a node is its tag string and attributes. Rename operation changes the node to another node.

The tree edit distance problem has been shown to be NP-hard \[95\] and many algorithms \[53, 77, 78, 89, 96, 112\] have been given to compute the tree edit distance. However, tree edit distance algorithms allow cross leveling which can break the structure of DOM trees. For example, a node can be the siblings of its parents after deleting its parent. Selkow et al. \[90\] address the issue and propose a simple tree matching algorithm. This algorithm keeps the order among siblings after processing. It matches nodes of two trees if the two nodes have the same label and maximizes the number of matched nodes. We summarize this simple tree matching algorithm in Algorithm 1.
The algorithm fills an $m \times n$ matrix $MATCH$, where $m$ and $n$ are the number of first-level sub-trees of two trees to be matched $T_1$ and $T_2$, respectively. The algorithm processes the two trees recursively. If the labels of roots are the same (cf. Line 1 in Algorithm 1), the algorithm then processes the first-level sub-trees. $MATCH$ records the number of matched nodes: $MATCH[i][j]$ is the maximum number of matched nodes when matching the first $i$ first-level sub-trees of the tree $T_1$ and the first $j$ first-level sub-trees of the tree $T_2$. The algorithm considers three cases when calculating $MATCH[i][j]$:

- The $j$th first-level sub-tree of tree $T_2$ is mismatched. The maximum number of matched nodes thus equals to $MATCH[i][j - 1]$ (cf. Line 10 in Algorithm 1). $MATCH[i][j - 1]$ is the maximum number of matched nodes when matching the first $i$ first-level sub-trees of tree $T_1$ and the first $j - 1$ first-level sub-trees of tree $T_2$.

- The $i$th first-level sub-tree of tree $T_1$ is mismatched. The maximum number of matched nodes thus equals to $MATCH[i - 1][j]$ (cf. Line 11 in Algorithm 1). $MATCH[i - 1][j]$ is the maximum number of matched nodes when matching the first $i - 1$ first-level sub-trees of tree $T_1$ and the first $j$ first-level sub-trees of tree $T_2$.

- The $i$th first-level sub-tree of tree $T_1$ and the $j$th first-level sub-tree of tree $T_2$ is
matched. The maximum number of matched nodes thus equals to $\text{MATCH}[i - 1][j - 1] + \text{Simple.Tree.Matching}(T_1(i), T_2(j))$ (cf. Line 12 in Algorithm 1), where $\text{Simple.Tree.Matching}(T_1(i), T_2(j))$ represents the maximum number of matched nodes of matching the $i$th first-level sub-tree of tree $T_1$ and the $j$th first-level sub-tree of tree $T_2$.

The algorithm selects the case described above that has the maximum number of matched nodes and records the number in $\text{MATCH}[i][j]$ after each step (cf. Line 9 in Algorithm 1). $\text{MATCH}[m][n]$ thus equals to the maximum number of matched nodes of tree $T_1$ and tree $T_2$. Besides, the matching nodes can be tracked by recording the operations in $\text{MATCH}$ and backtracking from $\text{MATCH}[m][n]$.

There exist many variants of the tree similarity algorithms described above, such as weighted tree matching [44] and weighted tree edit distance [98]. Compared with the unweighted versions, the weighted versions assign weights to node operations. Weights usually depend on the level of the node and the order among siblings. The weights are varying because nodes have different contributions to the change of structure. For example, operations on the root may change the structure more significantly than those on a leaf node.

The tree similarity algorithms described before (1) are easy to implement, (2) maintain the order among siblings, (3) not only calculate the similarity (e.g., calculate the number of matched nodes) but also align the trees (e.g., find the matched and mismatched nodes). These advantages make the tree similarity algorithms be widely used in web information extraction. However, there are drawbacks to these methods, which are listed below:

- The cost of editing or matching nodes is fixed.
- They only perform local optimization.

For the first drawback, although some algorithms allow different weights for different nodes, the weights are fixed for every step. The change of structure after every step of matching does not reflect by the change of weights. For the second drawback, most
previous work only performs local optimization when attempting to find the maximum number of matched nodes. The simple tree matching algorithm described above is an example.

We address these problems and give a global alignment algorithm which allows the costs (weights) to change after every step of matching. The global alignment algorithm processes trees recursively in a bottom-up manner and aligns the first-level sub-trees globally. The global alignment algorithm allows the costs (weights) of node operations to be different for every node and automatically updates the costs (weights) of node operations when the structure of the sub-tree changes.

2.3 Template Detection

Template detection techniques construct a wrapper to extract data records. A wrapper \[33,55\] is a pattern that describes the features of data records in a web page. The features include HTML tags, index among siblings, etc. Once a wrapper is constructed, it can be used to extract data records from web pages which are generated by the same template. A wrapper can be described in many ways, such as regular expressions and XPath. We first review regular expression based techniques in Section 2.3.1 and XPath based techniques in Section 2.3.2. We then describe techniques using variants of the DOM tree as the wrapper in Section 2.3.3. We also review some techniques that are designed for extracting data records from a set of web pages in Section 2.3.4.

2.3.1 Regular Expression Based Techniques

A regular expression \[97\] is a sequence of characters that describe a pattern. It is a common method to represent patterns in semi-structured text and unstructured text. Regular expression based techniques usually utilize the HTML tag string and semantics of HTML tags in rules to construct regular expressions. For example, the path from a table’s root (\(\langle table\rangle\)) to its cells (\(\langle tr\rangle\)) corresponds to a regular expression (\(\langle table\rangle \langle tr\rangle (.*?\langle tr\rangle \langle table\rangle)\)).
Regular expression based techniques can be classified into unsupervised techniques and supervised techniques. Unsupervised regular expression based techniques construct regular expressions using the HTML tags that wrapped the data records. The regular expressions are utilized to find all strings in the web pages that are matched and extract the data records from the matched strings. However, regular expressions are vulnerable, which may become invalid when web pages change. Besides, creating a suitable regular expression is time-consuming. Further, the flexibility of HTML is difficult to cope with by regular expression based techniques. For example, ⟨a⟩⟨/⟩⟨p⟩ and ⟨a⟩⟨/⟩⟨a⟩⟨p⟩⟨/⟩⟨p⟩ have the same meaning. However, they require different regular expressions to recognize.

Regular expressions constructed by unsupervised techniques depends on the features and rules selected by users. The features usually include HTML tags, the format of data records, etc. The rules depend on the selected features. However, some useful features and rules may be lost due to the selection of users. To address this issue, many supervised techniques \cite{39,61,62} have been proposed. Li et al. \cite{62} propose a transformation algorithm, named ReLIE, for learning complex regular expressions. It can learn patterns of entities, such as phone number and email, from real datasets. However, it is not designed for web information extraction, which makes it hard to learn the patterns from web pages. To solve the problem, Bex et al. \cite{8} work on learning deterministic regular expressions from tree-structured documents. However, as web page development techniques have changed dramatically during past years and supervised techniques usually need plenty of labeled training data, supervised regular expression based techniques are usually expensive and lack extensibility.

We use the following representative study to elaborate how regular expression based techniques work. Sahuguet et al. \cite{87} propose W4F which utilizes the HTML syntax to construct regular expression patterns. W4F consists of a retrieval layer, an extraction layer, and a mapping layer. It allows users to mark boundaries of data records on web pages directly with the help of a graphical user interface \cite{75}. The retrieval layer records the boundaries of marked records with invisible annotations on a web page. Users can
click the data records and add a label to the data record. Then, the extraction layer extracts regular expressions from the labeled data using rules. For example, one rule is that find the common nodes of paths from the root of the DOM tree of the web page to the data records. After extracting regular expressions, the extraction layer utilizes regular expressions to find all matched data records in the same web page. Finally, the mapping layer maps the extracted data records to labels given by users.

Regular expression based techniques usually utilize features based on DOM and process the content of a web page as a string. The regular expressions can be reused on web pages generated by the same template. It is efficient when processing a large number of web pages because of the efficiency of regular expressions. However, the regular expression based techniques have drawbacks which are shown as follows:

- The performance of regular expression based techniques depend on the quality of the labeled data, because regular expressions are extracted from labeled data. If the labeled data cover more possible cases of the data records, the regular expressions may match more strings and are likely to extract data records correctly. Otherwise, many data records may be missed because they cannot be matched with the regular expressions. The extracted data records may contain noise when the regular expression is too simple.

- Regular expression is vulnerable and becomes invalid when the web page changes, which brings extra challenges for the maintenance of regular expressions.

- The recognition of web components that contain data records is achieved by labeled data. The web components that contain data records are recognized manually and reflected in the labeled data. As data labeling is usually expensive, and the adaptation of regular expression based techniques is limited.

We address the problems described above and develop a splitting algorithm which can recognize web components generated by the same template. The splitting algorithm splits web components into chunks, and the web components in the same chunk are generated by the same template. This is achieved by generating the templates by differ-
ent splitting plans and select the best splitting plan. Besides, to tackle the vulnerability of
the wrapper (regular expressions), we represent the template by the layout tree which is
a variant of the DOM tree. The layout tree is robust and efficient to deal with changes to
web pages.

2.3.2 XPath Based Techniques

In this section, we describe XPath based techniques. XPath is designed for addressing
elements of tree structure documents (e.g., HTML, XML, etc.). It is more flexible than the
regular expression when representing the template of web pages, which makes the XPath
based templates more robust.

XPath is based on XML [12], which is an extensive markup language. W3C standardizes
XML [21] [25] to represent tree structure documents. XPath stands for XML Path Language
and is designed to locate elements of tree structure documents. Although XPath is not
specially designed for representing templates of web pages, the flexibility of XPath can
help locate data records in the web page. Zhang et al. [111] propose a data record ex-
traction method which utilizes XPath pattern learning to detect templates in a web page.
XPath pattern learning extracts the data records with similar XPath and considers them
having similar semantics. Okada et al. [71] and Gulhane et al. [39] adapt XPath to ex-
tract data from a set of web pages and find the relationships among the set of web pages.
Compared with templates written in regular expressions, templates written in XPath are
less influenced by the changes of web pages. Besides, XPath templates are effective to
locate elements in web pages which increase the efficiency of creating templates.

We further describe a representative work, named XPath pattern learning [111], for ex-
tracting data records from a set of web pages. It creates wrappers written in XPath and
utilizes the wrappers to extract data records in web pages generated by the same tem-
plate. They define a concept, named XPath pattern, to describe a sequence of location
steps. The location steps are the paths from the root of a DOM tree to a text node. They
develop a heuristic algorithm to map source paths to destination paths. The source path
is the XPath from the root to the data record in a web page, and the destination path is in the format of \( X/P/R \), where \( X \), \( P \) and \( R \) are nodes. In particular, \( R \) is the node that contains data record; \( P \) is the parent node of \( R \); \( X \) are nodes from the root of the DOM tree to the parent of \( P \). The algorithm maps every path in the DOM tree of a web page to the destination path. The mapping types are shown as follows:

- Single mapping, one to one.
- Sequential mapping, one to one.
- Choice mapping, many to one.
- Aggregation mapping, many to one.

![Diagram](image-url)

Figure 2.2: XPath pattern mapping
To describe the four types of mapping, we give an example shown in Figure 2.2. A, B, C, D, E are nodes in a DOM tree of a web page. Mapping A/B/C, in which C does not have siblings, to the destination path (X/P/R) is a single mapping (cf. Figure 2.2(a)). In Figure 2.2(b) as B has siblings. The sequence of C among their siblings is used to identify, for example, the C is the first child of B. Therefore, mapping such a A/B/C to X/P/R is a sequential mapping. Besides, as X/P/R only has one type of path (A/B/C), the sequence mapping is a one to one mapping. In contrast to the single mapping, when the children of B maps to R (cf. Figure 2.2(c)), mapping A/B/C to X/P/R is an aggregation mapping. The aggregation mapping is a many to one mapping because of R maps from different nodes (C, D and E) while X and P maps from the same node (A and B respectively). If the index of B’s children that is mapped to R is different, mapping A/B/(C/D) to X/P/R is a choice mapping. Choice mapping is also a many to one mapping because the R refers to different children of B.

To choose the best XPath pattern, Zhang et al. [111] define a distance named XPath distance, which consists of tag distance, expression distance, and node distance. They are used to measure the distance for mapping nodes in source paths to XPath patterns. Zhang et al. propose an algorithm to construct all the mappings and calculate the XPath distance of all mappings. The algorithm select the minimum distance mapping as the best mapping.

The XPath based techniques are flexible to record different paths and more robust than regular expression based techniques. Besides, they can automatically detect the template of a web page and recognize similar templates whose XPath distance is short. However, XPath based techniques have the following drawbacks:

- The factors of XPath distance is fixed for every node. The XPath distance consists of three factors: tag distance, expression distance, and node distance. The factors are fixed and hard to add new factors because the XPath does not record other information.
- Different XPath patterns cannot be aligned. The mappings are fixed and hard to
convert one mapping into another mapping by alignment.

To address the above problems, we develop an alignment algorithm which can utilize any self-defined factors. This is achieved by evaluating the alignment via an optimization function. The optimization function can consist of self-defined factors. For example, the maintainability index function considers operator, operands, etc. Besides, our alignment algorithm lifts the limitations of the format of the template. The template does not have a fixed format (four mappings as described above). The alignment algorithm can align templates (the layout tree) to generate a generalized template that covers all parts of all templates.

### 2.3.3 Tree Based Techniques

We discuss tree-based techniques in this section. Tree-based techniques usually create wrappers using variants of the DOM tree. The variants of the DOM tree can keep the tree structure, which is more robust in comparison with the regular expressions and the XPath. With the tree-based wrappers and tree similarity algorithms, tree based techniques can automatically recognize the web components generated by the same template. If the tree structure similarity of two sub-DOM trees is bigger than a threshold, two sub-DOM trees are considered as being generated by the same template. The main steps of tree-based techniques are:

- **Segmentation** Segment areas, which contain data records, from a web page.

- **Alignment** Align the data records in different areas using selected features, such as visual cues and tree structures.

The segmentation step locates the web components in a web page that contains data records. The web components containing data records are likely to be generated by the same template. The alignment step aligns the data records in the web components to construct structured data records. Many tree based techniques consist of segmentation and alignment. Liu et al. [64] use tree structure similarity to recognize repetitive web components, which contain data records in a web page. They do not align the data records. The
extracted data records thus do not include the structural information. Zhai et al. \cite{109} propose an improved method, named partial tree alignment, to align data records in different sub-DOM trees. The alignment algorithm captures the structure of data records. The alignment of data records is also named schema deducing \cite{49}. It aims to deduce the schema that provides the data records. To improve the performance of the alignment algorithm, Zhai et al. \cite{110} further apply the partial tree alignment algorithm to extract data records from a web page. One drawback of the work is that the alignment step is based on the result of the segmentation step. They assume that the web components to be aligned are generated by the same template. The alignment step thus cannot perform well when the web components are not generated by the same template. To tackle this issue, the state-of-the-art algorithms \cite{49,73} integrate the segmentation step and alignment step to recognize the web components generated by the same template and extract data records from these web components at the same time.

We give a representative work named partial tree alignment \cite{109} which is inspired by the simple tree matching algorithm. It inserts the first-level children of one tree into another tree only when the insertion position can be uniquely determined. Given two tree $T_1$ and $T_2$ whose roots have the same HTML tags, inserting the tree $T_1$ into the tree $T_2$ has three possible cases:

- The rightmost node of the first-level children of tree $T_1$ is the same as the leftmost node of the first-level children of tree $T_2$.
- The leftmost node of the first-level children of tree $T_1$ is the same as the rightmost node of the first-level children of tree $T_2$.
- Two consecutive nodes of the first-level children of tree $T_2$ are in the first-level children of tree $T_1$, and one of the nodes is the boundary node of the first-level children of tree $T_1$.

As shown in Figure \ref{fig:tree_alignment}, we denote the tree before “+” as $T_1$ and the tree after “+” as $T_2$. The tree after “=” is the tree after insertion. In the first case, the rightmost node $C$ in $T_1$ is the leftmost node in $T_2$, other nodes ($D$ and $E$) of $T_2$ are inserted after $C$ and the result is
2.3 Template Detection

(a) Unique expansion 1

(b) Unique expansion 2

(c) Unique expansion 3

(d) Insertion ambiguity

Figure 2.3: An example of the partial tree alignment
In the second case, the leftmost node A of $T_1$ is matched with the rightmost node A of $T_2$, the other nodes (D and E) of $T_2$ are inserted in the left side of A, the result is E, D, A, B, C (cf. Figure 2.3(a)). For the third case, two nodes (B and D, cf. Figure 2.3(c)) are matched and they are consecutive in $T_2$. The other nodes are inserted into the right of D. The insertion result is A, B, C, D, E. In Figure 2.3(d), there is a node C between B and D in tree $T_1$ and a node E between B and D in tree $T_2$. The order of C and E are not fixed when inserting the first-level children of tree $T_2$ into the first-level children of tree $T_1$, it results in insertion ambiguity. As the partial tree alignment algorithm only inserts the nodes into the leftmost side or the rightmost side of another tree, the alignment is named partial alignment. The main idea of aligning a forest using the partial tree alignment algorithm is that select one tree as the template and insert all the other trees into the template. The pseudo code of the partial tree algorithm is shown in Algorithm 2.

**Algorithm 2: Partial Tree Alignment**

**Data:** $F$ is a forest of trees

1. Sort $F$ by the number of unaligned data records in descending order
2. $not\_aligned = false$
3. $F_{imp} = []$
4. $inserted = False$
5. $T_{seed} = Shift(F)$
6. **while** $F$ is not null **do**
7. $T_c = Shift(F)$
8. align_trees($T_{seed}, T_c, \text{Simple\_Tree\_Matching}(T_{seed}, T_c))$
9. **if** $T_c$ is not completely matched with $T_{seed}$ **then**
10. $inserted = InsertT_c$ into $T_{seed}$
11. **if** inserted is false **then**
12. $Insert T_c$ into $F_{imp}$
13. **if** inserted is true and Simple_Tree_Matching($T_{seed}, T_c$) is not null **then**
14. $not\_aligned = true$
15. **if** $F$ is null and not_aligned = true **then**
16. $F = F_{imp}$
17. $F_{imp} = []$
18. $not\_aligned = false$
19. $inserted = False$
20. **return** $T_{seed}$
2.3 Template Detection

The partial tree alignment algorithm first sorts the forest ($F$) by the number of unaligned records in descending order (Line 1). It then shifts the first tree by $F$, which is denoted it by $T_{seed}$, in Line 2. It inserts following trees in $F$ into $T_{seed}$ (Line 3). From the first tree of $F$, denoted by $T_c$ (Line 4), the algorithm tries to match $T_c$ with $T_{seed}$ based on the results of simple tree matching algorithms (Line 8). Line 9-12 shows that if $T_c$ and $T_{seed}$ cannot be completely matched, it tries to insert $T_c$ into $T_{seed}$. If not all unaligned items in $T_c$ are inserted into $T_{seed}$, $T_c$ is left for further processing. Line 15 checks that if all trees in $F$ are processed, and all alignments and insertions are completed. It then processes the trees that are stored in $F_{tmp}$ and inserts them into $T_{seed}$.

Zhai et al. [110] apply the partial tree alignment algorithm on record-level information extraction and present a framework, named DEPTA. It utilizes visual cues to segment web pages. This step aims to identify data records on the web page. Then, it aligns the sub DOM trees containing data records and extracts data records from the web page. Based on the visual cues (e.g., height, width, color) and tree similarity algorithms, it locates some regions on web pages, which contains similar content (e.g., tables and lists). The region is named data region. It utilizes the partial tree alignment algorithm to align the sub DOM trees of these regions and extract data from them.

By exploiting the power of visual information and tree similarity algorithms, DEPTA is a powerful framework and performs well in data record extraction. However, with the development of HTML, especially the introduction of HTML5 [43], many websites utilize tables to format the layout of web pages. This breaks the assumption of visualization based techniques that tables and lists are used to wrap data records. This impinges a dramatic drop in the performance of the DEPTA. Besides, the partial tree alignment algorithm limits the accuracy of data record extraction because of partial alignment. The algorithm cannot process the insertion ambiguity cases.

The problems that we address in the partial tree alignment algorithms are summarized as follows:
• Web page segmentation relies on visual cues which do not always work.

• The partial alignment algorithm does not work when the insertion is not fixed.

Comparing with techniques using partial tree alignment, our work aligns first-level sub-DOM trees globally. Besides, as partial tree alignment algorithms utilize the simple tree matching algorithm, the cost of matching is fixed. Our alignment algorithm processes web pages with varying costs. Moreover, to improve the segmentation algorithm, we develop a splitting algorithm and using an optimization function to determine the similar regions on a web page. Furthermore, the segmentation and alignment work independently in most of the previous work. We propose segmentation algorithms (named splitting algorithms in our work) are based on the results of alignment.

2.3.4 Page-level Information Extraction

Web information extraction techniques can be classified by input: record-level information extraction and page-level information extraction. Record-level information extraction takes a single web page as the input and extracts data records in it. We focus on the latter type of work in this section. Page-level information extraction takes a set of web pages as the input. A set of web pages can provide much information (e.g., visual information, dom trees) for template detection. As the structures of many DOM trees are more complex than one DOM tree, this brings extra challenges for identifying data records and aligning data records.

Crescenzi et al. [30] present RoadRunner that utilizes a regular expression based method to recognize differences among the input web pages. RoadRunner considers one of the web pages as the template and aligns other web pages with the template. The web page selected as the template is expected to be complex enough to have all possible structures of sub-DOM trees of all DOM trees. This makes the template selection time-consuming and needs to be done manually, which is less practical. Kayed et al. [49] propose an approach, named FiVaTech, which gives a new structure called “fixed/variant pattern tree” to represent templates. They give a definition of page-level record extraction as follows:
Definition 21. **Page-level Data Extraction** Given a set of DOM trees, $\text{DOM}_i = \lambda(T, x_i)$, created from some unknown template $T$ and data records $x_1, \ldots, x_n$, deduce the template, schema, and data records from the set of DOM trees.

FiVaTech applies techniques, such as tree alignment and pattern mining, to extract data records from web pages and deduce the schema that provides the data records. The schema is the structural information of the data records. The data records in the schema are aligned by the attribute of each data record. For example, “apple” and “banana” are aligned because they are fruit, and “$1.5$” and “$8.0$” are aligned because they are the price.

TEXT [51] automatically extracts templates from a set of web pages. It detects the web components generated by the same template via measuring the minimum description length [84], which is the length of the code block of web components.

Once the template is extracted, the template should be verified because the noise in web pages affects the extraction of the template. Chang et al. [16] propose a method for validating templates extracted by unsupervised template extraction methods. They define the verification problem as a constraint satisfaction problem and verify the assignment of leaf node labels. They construct a finite state machine to provide constraints from previously learned templates.

We further detail FiVaTech [49] to show the workflow of page-level data extraction techniques. The workflow of FiVaTech is shown in Figure 2.4. FiVaTech converts web pages into DOM trees and merges similar DOM trees by aligning them. FiVaTech then extracts schemas and data records from the merged DOM trees. The core part of FiVaTech is the multiple tree merging which consists of four steps: peer node recognition, peer matrix alignment, pattern mining, and optional node merging. The multiple tree merging algorithm fills a matrix $M$, where every column stores the first level subtrees of all DOM trees. Therefore, every element of $M$ stands for a subtree. The multiple tree merging algorithm begins with recognizing peer nodes among all sub-trees. Peer nodes are sub-trees which
are similar to each other. Then, the peer nodes are aligned, especially the text nodes. Next, the multiple tree merging algorithm extracts patterns from the aligned peer trees and finally detect the optional nodes. An optional node is a node which is not shown in every column. The nodes shared by all sub-trees and the optional nodes composite the template of all web pages.

Peer tree recognition calculates the tree matching score, which is the similarity between two trees. Tree matching score is based on the result of the tree matching algorithm (described in Section 2.2.2). We summarize the algorithm by the pseudo code shown in Algorithm [5]. If the type of two nodes are not the same or the tag strings are different, they cannot be matched, and the tree matching score is 0. If two nodes are leaves or the number of nodes in two sub-trees are the same, the tree matching score equals to \( \frac{2 \times \text{TreeMatching}(T_1, T_2)}{\text{size}(T_1) + \text{size}(T_2)} \) (Line 4), where \( \text{TreeMatching}(T_1, T_2) \) equals to the number of matched nodes of the tree \( T_1 \) and the tree \( T_2 \). Otherwise, the tree matching scores equals to the average tree matching score of its children \( \left( \frac{\text{totalScore}}{\# \text{ first level subress of } T_1} \right) \) plus \( \frac{2}{\text{size}(T_1) + \text{size}(T_2)} \).
2.3 Template Detection

temporary score of matching sub-trees \( t_1 \) and \( t_2 \) equals to \( \frac{2 \times \text{TreeMatching}(t_1, t_2)}{\text{size}(t_1) + \text{size}(t_2)} \) (Line 10). If sub-tree \( t_1 \) and sub-tree \( t_2 \) can be matched (\( \text{tmp} \) is not zero), we add the temporary score to the total score (Line 12) and add one to the matched sub-trees (Line 13). The size of a DOM tree is the number of nodes. The total score is the average matching score of children (Line 14 - 16).

---

**Algorithm 3: TreeMatchingScore(\( T_1, T_2 \))**

**Data:** Two DOM trees \( T_1 \) and \( T_2 \)

1. if \( T_1 \) and \( T_2 \) have different root tag string then
   2. return 0;
3. if \( T_1 \) or \( T_2 \) is a leaf or \( \text{size}(T_1) \) equals to \( \text{size}(T_2) \) then
   4. return \( \frac{2 \times \text{TreeMatching}(T_1, T_2)}{\text{size}(T_1) + \text{size}(T_2)} \),
5. totalScore = 0;
6. for every first level subtree \( t_1 \) in \( T_1 \) do
   7. score = 0;
   8. matching = 0;
   9. for every first level subtree \( t_2 \) in \( T_2 \) do
   10. \( \text{tmp} = \frac{2 \times \text{TreeMatching}(t_1, t_2)}{\text{size}(t_1) + \text{size}(t_2)} \); 
   11. if \( \text{tmp} > 0 \) then 
       12. score += \( \text{tmp} \); 
       13. matching += 1;
   14. if \( \text{matching} > 0 \) then 
       15. score = \( \frac{\text{score}}{\text{matching}} \); 
16. totalScore += score;
17. return \( \frac{\text{totalScore} \times \text{first level subtrusses of } T_1}{\text{size}(T_1) + \text{size}(T_2)} + \frac{2}{\text{size}(T_1) + \text{size}(T_2)} \)

---

After recognizing peer nodes, the matrix alignment step begins with giving peer nodes the same symbol (e.g. “a” stands for “⟨div⟩”, “b” stands for “⟨img⟩”, etc.). Then, FiVaTech aligns the matrix row by row with aims that every row’s nodes have the same symbol. We give a running example shown in Figure 2.5 to describe the matrix alignment step.

The matrix alignment processes the matrix of nodes from left to right in a top-down manner. The matrix alignment step scan the node in the same row and locate the first node with a different tag from its previous node. For example, the third column of the first row is different (“b” is different from “a” in step (1) in Figure 2.5). The matrix alignment then shifts the column to align the node with others that contain the same symbol. For
example, the shift length is 1 in step (1) in Figure 2.5, aligning “b” with the second row. It repeats the detection and shifts until every row contains the same symbol. For the running example in Figure 2.5, it first shifts the first row of the third column (“b”) and the shift length is 1 (the step (1)). It then shifts the third row of the second column (“d”) and the shift length is 1, which results in the step (3). Finally, it shifts the “e” down and results in the aligned children which are “a/b/c/d/e”.

After aligning children of different trees in the last step, FiVaTech mins patterns from the aligned children. For example, if the aligned children is “a/b/c/a/b/c/d”, the pattern should be “a/b/c/d”. After the mining step, FiVaTech detects optional nodes (e.g. “d” in “a/b/c/d”). Finally, the template of the DOM tree is extracted, and FiVaTech constructs the schema from the template.

FiVaTech can extract the template and the structure of data from web pages. However, it cannot maintain the sequence of siblings. The matrix alignment step considers the tree which has the maximum number of children as the template and aligns other trees with the template. This breaks the continuation of siblings because the aligned two trees may not be continuous siblings. Besides, the peer node recognition, alignment, and pattern mining are independent. Two nodes which are not peer nodes may be aligned. Considering the “a/b/c/a/b/c/d” example given above, two “a” may not be peer nodes but are aligned because they have the same tag. The pattern mining algorithm may reduce
the accuracy of extracting data records.

We address the following problems from FiVaTech:

- The order among siblings cannot be maintained after alignment.
- The pattern of nodes may record peer nodes incorrectly.

To address the problems given above, we align the siblings and recognize the peer nodes at the same time. The layout tree generated by aligning sub-trees is the template of sub-trees. This can avoid the incorrect records of peer nodes because the template is constructed according to the recognition of peer nodes.

Besides, the page-level data extraction assumes that the set of web pages are generated by the same template. We lift the assumption and aim to extract data records from a set of web pages which are generated by different unknown templates.

2.4 Software Maintenance

In this section, we briefly present conventional software maintenance techniques and discuss their relationships with our work.

If there exists a code block which is similar or identical to other code blocks, this case is named as code clone [46]. The cloned code can be simplified by adding loops. The code block after being processed is like the template code of cloned code blocks. We thus present techniques for detecting code clone. Code clone is harmful to software maintenance because it increases the operands, operators, lines of code and decision sentences in programs. Kamiya et al. [46] develop a tool, named CCFinder, to detect code clone in programs which are written in the C programming language. It utilizes techniques, such as lexical analysis and format transformation, to detect code blocks which have the same logical structures and outputs. However, users have to develop new lexical tools when adapting CCFinder to programs written in other programming languages. As CCFinder
is specially designed for C programming languages, it utilized features and suggested format of C language programs, which makes it does not perform well in analyzing other languages. Wahler et al. [101] propose a method to detect code clone via the frequency of items. It analyses the keywords of a program and formats them into an XML file. Every keyword is replaced with a label (e.g., while, for, do-while are replaced with “⟨loop⟩”). It then finds the items with the same label and analyses the frequency to detect clone code. The work is vulnerable because programs with different structures are considered as unique programs even if they have the same function. Venkatasubramanyam et al. [99] focus on large code bases and provide a systematic method to detect clone code and determine the order of fixing. Roy et al. [86] evaluate tools on code clone and provide practical suggestions for choosing appropriate tools. Keasler et al. [50] give a tool named TALC which focuses on the structure of data and improves the maintenance of program via reducing the repetitive variables.

There are issues in extracting easily maintainable template code via conventional techniques as follows:

- Different lexical analysis tools are required for different programming languages. Conventional methods need researchers to develop different tools for different programming languages, which is expensive for web page development because there are many programming languages and libraries, such as Angular Js, ASP, JSP, PHP, etc.. Software engineers are likely to work in different programming languages and software maintenance is difficult if various tools are used.

- The clone code detection is based on the code of a program instead of the output. A programmer can write many blocks which do not affect the output because of the input data. For example, a programmer can write a “if-do” sentence which consists of many lines of code while the decision never triggers. This brings extra challenges for researchers because they have to develop tools that particularly detect the unused code blocks.

To address the issues above, we extract easily maintainable template code from web page.
Programmers can find the unused code blocks by comparing with the extracted template code. Besides, we give a conception layout tree (detailed in Section 3.2) which can be easily translated into different programming languages and libraries.

2.5 Summary

We reviewed the related literature on both web information extraction and software maintenance. Although previous work has given inspiring references, none of them can be directly applied to solve our problem. This motivates our studies in the following chapters.
Chapter 3
Web Page Separation for High-quality Template Code

3.1 Introduction

Separating a web page into template code and data records populated into the template is an important problem. This problem has a wide range of applications in web page compression and data record extraction. Besides, separating a web page in lossless manner can reconstruct the original web page with extracted template and data records. Compared with data record extraction [1, 16] which discard the template of web pages, web page separation keeps the template of web page, which can be useful for further processing, such as template code analysis. We study this problem and focus on the maintainability of the template code generated in this chapter, since easily maintainable template code is reliable, and it will simplify further developments on top of the template code, e.g., to update the web templates. Figure 3.1 shows an example of the web page separation problem. In particular, Figure 3.1(a) shows the HTML source of a web page which contains a list of items (cf. the “⟨li⟩” tags). Figure 3.1(b) shows the template code separated from the HTML source, which effectively says that the HTML page can be generated by a for-loop (cf. the “⟨for1⟩” tag) to produce a list of “⟨li⟩” items. Variables “r1 represents the attributes and data records to be populated into this list of items” and “r2”, the values of which are stored in a data record file as illustrated in Figure 3.1(c). Note that, in this

\(^1\)The main content of this chapter has been accepted for publication in the following paper: Chenxu Zhao, Rui Zhang and Jianzhong Qi, “Web Page Template and Data Separation for Better Maintainability,” International Conference on Web Information Systems Engineering (WISE), 2018.
example, HTML source code, the second list item contains an additional "〈span〉" tag. This is reflected in the "〈i f1〉" tag in the generated template in Figure 3.1(b).

```html
<div>
  <li><a href="java">Java</a></li>
  <li><a href="php">PHP</a></li>
  <span>STAR</span></div>

(a) Web page source

```html
<div>
  <for1>
    <li>
      <a href="{r1}">{r2}</a>
      <if1><span>STAR</span></if1>
    </li>
  </for1>
</div>

(b) Template code

```

```html
  {r1:java,r2:Java},
  {r1:php,r2:PHP,if1:[]}
```

(c) Data records

Figure 3.1: An example of the web page separation problem

Making such a separation has important applications. For example, we can improve the maintainability of web page source code by writing "for-loop" to produce repeating data records rather than writing a duplicate code segment for each data record. Besides, we may reduce the size of the web page by storing it in its template code form, e.g., a long list in the HTML source can be replaced by a simple "for-loop" tag in the template code.

SYNTTHIA \[73\] is the state-of-the-art web page separation algorithm that aims at compressing web pages. It processes the DOM tree of a web page hierarchically and utilizes pairwise similarity to determine which siblings should be folded, i.e., replaced by a for-loop in the template code. It uses a local alignment algorithm to capture the differences between the siblings folded together, i.e., to add the "if" tags in a for-loop in the template code. After separating a web page into its template and data records, SYNTTHIA will use...
the template code and data records to replace the HTML source only if they are shorter. For example, the number of characters of the HTML source in Figure 3.1(a) is 90, while the total number of characters in the template code and data records in Figures 3.1(b) and 3.1(c) is 131. Thus, SYNTHIA will keep the HTML source as it is shorter.

In comparison, our work focuses on the maintainability of the web page source. In the example shown in Figure 3.1, the HTML source has a maintainability score of 110.106, while the template code has a maintainability score of 110.132 which indicates that the template code is easier to maintain (detailed in Section 3.2.3). Intuitively, replacing all those list items with a for-loop makes it easier to update the web page, e.g., if we want to change the style of each list item, we only need to change it once in the template for-loop rather than changing every list item in the HTML source.

In this chapter, we study how to separate a web page into easily maintainable template code and data records. We evaluate the maintainability of a separation by the maintainability index (MI) \[27\]. MI is an important software metric to measure the maintainability of software source code. A higher MI score suggests higher maintainability and is more preferable. Intuitively, a piece of code would have a higher MI score if it is shorter, having fewer variable and functions, and having fewer branches (detailed in Section 3.2.3). To the best of our knowledge, we are the first to formulate a web page separation problem based on the MI score. We analyze the complexity of our separation problem and show that it is NP-hard. We then adapt the SYNTIA algorithm to solve our problem. In SYNTIA, the tree sibling splitting procedure compares pairwise sibling similarity and uses a splitting threshold to determine whether the siblings should be folded together or be separated into different chunks. This threshold is heuristically defined to obtain short template code. We adapt the algorithm by replacing this threshold with ours that aims to obtain a high MI score.

We further develop a population-based optimization algorithm, named dual teaching and learning based optimization (dual-TLBO), to optimize sibling splitting. The algorithm con-
siders a splitting plan as an individual and evaluates the maintainability of a population of different individuals (splitting plans) globally under the subtrees being considered, which addresses the limitation of SYNTHIA that only considers pairwise similarity. We also develop a global alignment algorithm with dynamic costs of alignment operations, which achieves a high-quality separation. In summary, we make the following contributions:

- We define a novel web page separation problem and show that it is NP-hard.
- We propose a population-based algorithm named dual-TLBO to help select the siblings in the web page source code tree that should be folded together to achieve a high MI score.
- We propose a global alignment algorithm to align siblings to achieve a high MI score.
- We perform an experimental study to evaluate our algorithms in both maximizing the MI score and minimizing the length of the generated template code and data records.

The rest of this chapter is organized as follows: Section 3.2 presents basic concepts and baseline algorithms. Section 3.3 defines the problem and analyzes its complexity. Section 3.4 details the splitting algorithm, the folding algorithm and the alignment algorithm. Section 3.5 presents experimental results. Section 3.6 concludes this chapter.

3.2 Preliminaries

In this section, we first briefly discuss an existing algorithm called SYNTHIA [73] in Section 3.2.1. We utilize the conceptions of SYNTHIA to define our problem and describe our heuristic algorithm. We then describe an optimization algorithm named teaching and learning optimization (TLBO), in Section 3.2.2. We improve the TLBO and make it select good solutions in element-level. We also describe a software metric for maintainability, the maintainability index, in Section 3.2.3.
3.2 Preliminaries

3.2.1 SYNTHIA

SYNTHIA is an algorithm that separates a web page into a template component and a data component. It aims to minimize the *minimum description length* (MDL) of the template component and the data component. Here, the MDL is the length of characters of the extracted template code and the extracted data records. SYNTHIA starts with parsing the web page into a DOM tree. A *DOM tree* (cf. Figure 3.2(a)) is an ordered and rooted tree which consists of element nodes and text nodes. An element node (e.g., the “⟨div⟩” node in Figure 3.2(a)) has a tag string (e.g., “div”) and a list of attributes. An attribute is a mapping from an attribute name to text. A text node (e.g., the “Java” node in Figure 3.2(a)) is a leaf node of a DOM tree whose value is text. The data component is named an *environment*. An environment (cf. Figure 3.1(c)) is a mapping from variables to values where the value is a text string or a list of environments. The template component is represented by the *layout tree*. A layout tree is a transformed representation of a DOM tree. For example, it has three types of *instruction nodes*: *condition*, *iteration*, and *reference*. An iteration node (e.g., the “(for1)” node in Figure 3.2(b)) represents a repetition of child nodes in the corresponding DOM Tree (e.g., the two “⟨li⟩” nodes in Figure 3.2(a)). A condition node (e.g., the “(if1)” node in Figure 3.2(b)) shows an optional child in the corresponding DOM tree (e.g., the “⟨span⟩” node in Figure 3.2(a)). The reference node (e.g., “[r1]” in Figure 3.2(b)) captures the difference of attributes and text nodes between siblings. For example, the values of the attribute “href” of the two “⟨li⟩” nodes in Figure 3.1(a) which are “java” and “php” respectively, are captured in “[r1]” in Figure 3.2(b).

A combination of a layout tree and an environment that can generate the original DOM tree is called a *separation*. The separation algorithm traverses the DOM tree bottom-up recursively. For siblings, it *splits* siblings to *chunks* and *fold* them into one layout tree. An iteration node, whose tag string is “for”, is added to instruct that it is generated by folding several siblings. The difference of tree structure is captured by adding a condition node, whose tag string is “if”. Also, the difference between attributes and text nodes between siblings are instructed by reference node. A reference node is a symbol that maps the variables in the layout tree to

The separation algorithm traverses the DOM tree bottom-up recursively. For siblings, it splits siblings to chunks and fold them into one layout tree. An iteration node, whose tag string is “for”, is added to instruct that it is generated by folding several siblings. The difference of tree structure is captured by adding a condition node, whose tag string is “if”. Also, the difference between attributes and text nodes between siblings are instructed by reference node. A reference node is a symbol that maps the variables in the layout tree to
Let us revisit the example shown in Figure 3.1. Due to the space limit, we hide the attributes of the DOM tree element nodes (e.g., “href” attribute for a hyperlink node). The layout tree in Figure 3.2(b) is separated from the DOM tree shown in Figure 3.2(a) by folding the two branches with “(li)” nodes. After folding, the layout tree has a root “(for1)”. The number “1” here is the node index number. The branch, whose root is “(span)”, is an optional branch. Therefore, a node “(if1)” is added to the layout tree. Further, “[r1]” refers to the attributes (“href”), and “[r2]” records text nodes. After alignment, the corresponding environment is shown in Figure 3.1(c). “for1”, “r1”, “r2” and “if1” of the environment refers to “(for1)”, “[r1]”, “[r2]” and “if1” of the layout tree respectively. Finally, SYNTHIA compares the number of characters of the HTML source of the web page with that of the template code and the environment and returns the shorter of the two representations as it aims to minimize the length of the web page representation.
3.2 Preliminaries

3.2.2 Teaching and Learning Based Optimization Algorithm

Teaching and learning based optimization (TLBO) [81] is a population-based optimization algorithm that has been shown to outperform many conventional optimization algorithms, e.g., Genetic Algorithm. It is inspired by how students learn from their teacher and peers towards solving an assignment.

The general workflow of a TLBO algorithm is shown in Figure 3.3. TLBO randomly generates pop_size individuals as the initial population. The current best solution is selected to be the teacher. In the teacher phase, TLBO computes the difference between the teacher and the mean of the population. If every individual is a tensor whose length is $l$, the population is a matrix whose size is $l \times pop\_size$. The mean of the population is computed column-wise. For every individual, TLBO computes a new individual and replaces the old individual $X_{old}$ if the new one $X_{new}$ (computed by adding the mean dif-
ference weighted by a hyperparameter $r$ to $X_{old}$ is a better solution. In the student phase, every individual learns from two randomly selected individuals and replaces the old individual with the winner between the old and new individual. Finally, TLBO checks the termination criterion which is application dependent and stops when the termination criterion meets.

3.2.3 Maintainability Index

The maintainability index (MI)[27] is a software metric to evaluate the maintainability of programs. We use it to evaluate the quality of separation. Three important software metrics form MI: Halstead Volume ($V$), Cyclomatic Complexity[66] ($G$), and Source Lines of Code (SLOC). Let $\eta_1$ be the number of distinct operators, $\eta_2$ be the number of distinct operands, $N_1$ be the number of operators, and $N_2$ be the number of operands, the Halstead Volume $V = (N_1 + N_2) \log_2(\eta_1 + \eta_2)$. The Halstead Volume reflects how difficult it is to read a program, i.e., a lower value of $V$ suggests that only a small number of operators and operands are used throughout the program, which is preferable. The Cyclomatic Complexity is the logical complexity of the program. Given a control flow graph of a program, let $E$ be the number of edges, $N$ be the number of nodes, $P$ be the number of connected components, the Cyclomatic Complexity $G = E - N + P$, which equals the number of branches of a program plus 1. For example, if a condition or a loop contains a single branch, it adds one to the Cyclomatic Complexity. SLOC is the number of effective lines of code in a program (i.e., blank lines are excluded). Finally, the function of MI is given as $MI = 171 - 5.2 \times \ln(V) - 0.23 \times (G) - 16.2 \times \ln(SLOC)$. The coefficients in the MI function come from a sequence of modifications[22,23,72]. The MI function that we used is believed to measure the maintainability of programs better than previous functions. It is widely used by many companies. For example, Microsoft utilizes the given MI function in its famous product Visual Studio.

We evaluate our work using the MI score and aim to generate a separation that has a high MI score (i.e., better maintainability). To compute the maintainability index score for separation, we define the maintainability index of layout trees as summarized shown
3.3 Problem Definition and Complexity Analysis

Every text node adds one operand and one SLOC. The text node is identified by its value. An element node adds one operator. Element nodes usually consist of a starting tag and an ending tag, such as ⟨li⟩ and ⟨/li⟩. A starting tag or an ending tag adds one SLOC, respectively. They are identified by its tag string. A condition/iteration node adds one operator, one Cyclomatic Complexity, and two SLOCs. A reference node adds one operand. It further adds one SLOC when it does not refer to attributes of element nodes.

3.3.1 Problem Definition

We parse the web pages into a DOM tree. There exist many possible separations of a given DOM tree as formed by different folding combinations of the siblings in the DOM tree. We define a new concept called the variation to represent a specific folding combination. A variation of a given DOM tree is defined as a set \( S \) of subsets of siblings of the DOM tree that satisfies the following three conditions:

1. The intersection of any two subsets in \( S \) (i.e., subsets of siblings) is empty.
2. The union of all subsets in \( S \) contains all nodes of the DOM tree.
3. The siblings of an subset in $S$ are continuous, i.e., they are all adjacent to each other in the DOM tree.

The maintainability index of a variation of a given DOM tree is the maintainability index of the layout tree that is generated by the variation. To help understand the variation, we give a running example in Figure 3.4. We all nodes in the DOM tree (cf. Figure 3.4(a)) as an universe $X$, $X = \{E_1, E_2, E_3, E_4, E_5, E_6, E_7, T_1, T_2, T_3, T_4\}$. A possible variation of $X$ is $\{\{E_1\}, \{E_2\}, \{E_3\}, \{E_4\}, \{E_5\}, \{E_6\}, \{E_7\}, \{T_1\}, \{T_2\}, \{T_3\}, \{T_4\}\}$. The intersection of any two subsets in the variation is empty and the union of all subsets in the variation is $X$. As all subsets in the variation have one element, the variation correspond to a layout tree which is the DOM tree itself. Also, all the subsets in variations should be continuous ($\{E_5, E_6, E_7\}$ in Figure 3.4(b), $\{E_5, E_6\}$ in Figure 3.4(c) and $\{E_6, E_7\}$ in Figure 3.5(a)). The variation $\{\{E_1\}, \{E_2\}, \{E_3\}, \{E_4\}, \{E_5\}, \{E_7\}, \{T_1\}, \{T_2\}, \{T_3\}, \{T_4\}\}$ is invalid because $\{E_5, E_7\}$ is not continuous siblings in the DOM tree and folding them together cannot maintain the order of siblings when reconstruct the DOM tree from its corresponding layout tree and environment. Elements in every subset must be siblings. For example, $\{\{E_1\}, \{E_2\}, \{E_3, T_2\}, \{E_4\}, \{E_5\}, \{E_6\}, \{E_7\}, \{T_1\}, \{T_3\}, \{T_4\}\}$ is not a valid variation because $E_3$ and $T_2$ are not siblings and they cannot be folded.

We aim to find a variation that has the maximum maintainability index. A formal definition of our separation problem is as follows.

**Definition 31.** Maintainability based web page separation (optimization version) Given a DOM tree $T$ and the maintainability index function, find a variation that has the maximum maintainability index.

### 3.3.2 Complexity Analysis

To analyze the complexity of our separation problem, we first recast it as a decision problem as follows:

**Definition 32.** Maintainability based web page separation (decision version) Given a DOM tree $T$ and the maintainability index function, determine whether there exists a variation...
3.3 Problem Definition and Complexity Analysis

(a) A DOM tree

(b) A possible variation

(c) A possible variation
Then, we reduce the exact cover problem, which has been proved to be an NP-complete problem [48], to our separation problem. As the solution of a general exact cover problem does not have a weight (which is to be reduced to our MI score $c$), we define that all valid solutions of an exact cover problem have a constant weight $c^*$, while non-valid solutions have a negative infinity weight. The definition of an exact cover problem is as follows:

**Definition 33.** EXACTCOVER Given a set $X$, a set $S$ of subsets of $X$, and a cost function $F$, determine whether there is a subcollection $S^*$ of $S$ such that the intersection of any two distinct subsets in $S^*$ is empty, the union of the subsets in $S^*$ is $X$, and $F(S^*) = c^*$. Here, $F(S^*)$ is a function that returns a value of $c^*$ if $S^*$ is an exact cover of $X$, and $-\infty$ otherwise.

**Theorem 31.** The decision version of the maintainability based web page separation problem is NP-hard.

**Proof.** In the following, we construct an instance $ms$ of the decision version of the maintainability based web page separation and form a mapping from EXACTCOVER to $ms$. Given any $X$ of EXACTCOVER, we define the nodes of $T$ of $ms$ as the elements of $X$. A
subcollection of $S$ consists of many subsets of $S$, denoted by $S'$. If we view each subset of $S$ as a subset of siblings of $T$ in $ms$, then each subcollection corresponds to a variation in $ms$. We map each subcollection $S'$ that covers all elements of $X$ to its corresponding variation and define the cost of the variation as the cost of all subsets in $S'$. The cost of the variation corresponding to any other subsets of $X$ than those in $S$ is defined as $-\infty$. For any instance of $EXACTCOVER$, if there exists a subcollection of $S$ that covers all elements of $X$ and has the cost of $c^*$, then its corresponding variation also has the cost of $c^*$. Conversely, if there is a variation for $ms$ having the cost of $c^*$, this variation must correspond to a subcollection of $S$, since all other subsets of $X$ have the cost of $-\infty$. Therefore, $EXACTCOVER$ is reduced to the decision version of the maintainability based web page separation problem, and the decision version of the maintainability based web page separation problem is NP-hard.

### 3.4 Our Method

We propose the general procedure of our algorithm in Section 3.4.1. We then show the splitting algorithms, the folding algorithm and the alignment algorithm in Section 3.4.2, Section 3.4.3 and Section 3.4.4 respectively.

#### 3.4.1 Overall Algorithm Procedure

Our algorithm traverses the DOM tree in a bottom-up manner to generate a layout tree to maximize the MI score of every sub-layout tree. For every node in the DOM tree, our algorithm uses a splitting algorithm (i.e., dual-TLBO, detailed in Section 3.4.2) to split its child nodes (siblings) into chunks. For the chunks that contain multiple siblings, our algorithm folds them together and uses an alignment algorithm (detailed in Section 3.4.4) to capture the differences among the siblings. The splitting algorithm aims to maximize the MI score of the layout trees which are generated from the chunks. The folding algorithm adds iteration nodes (“for node”) for a multiple-siblings chunk, and the alignment algorithm adds condition nodes (“if”) to the layout tree generated to capture differences among siblings in the same chunk. When the root node of the DOM tree is reached, the
Algorithm terminates and returns the layout tree generated. Algorithm 4 summarizes the algorithm. In application, the folding algorithm and the alignment algorithm cooperate to convert a chunk into a layout tree.

Algorithm 4: SeparationAlgorithm(T)

Data: A DOM Tree T

1. if T is a text node then
   2. return T;
2. if T is an element node then
   3. if T.children is null then
      4. return T;
   5. for child_i in T.children do
      6. replace child_i with SeparationAlgorithm(child_i);
   7. chunks = Split(T.children);
   8. for chunk in chunks do
      9. if chunk.length > 1 then
          10. replace chunk with Fold_Align(chunk);
      11. replace T.children with chunks;
   12. return T;

We denote the DOM tree by T, and T.children denotes its first level subtrees. We name our general algorithm as SeparationAlgorithm. Our general algorithm first checks the type of T’s root and return T’s root without processing if it is a text node (Lines 1 - 2) or it is an element node without children (Lines 4 - 5) (e.g. “⟨/img⟩”). As the general algorithm works recursively, it replaces every first level subtree child_i with SeparationAlgorithm(child_i) in Lines 6-7. As the algorithm works in a bottom-up manner, SeparationAlgorithm(child_i) returns the layout tree of child_i, and the child_i is a layout tree whose root is an element node or a text node. The general algorithm then splits the first level subtrees (the layout tree of child_i) of T into chunks (Line 8). For every chunk, if the chunk has more than one subtree (Line 10), we fold the subtrees into one layout tree and aligns them to capture the differences. Every chunk becomes a layout tree. Line 12 replaces the first level subtrees of T with the layout trees that are generated from chunks. After folding and alignment, the root of layout trees can be an iteration node (when the chunk has more than one subtree), or an element node (when the chunk has one subtree whose root is an element node), or a text node (when the chunk has one or more subtrees whose root are text nodes).
To help understand our general algorithm, we next give a running example, in Figure 3.5. To help identify nodes with the same tag string, we mark them with its node type and index. For example, $E_1$ (“⟨div⟩”) stands for an element node whose index is 1, and $T_2$ (Bed) stands for a text node whose index is 2. Our general algorithm traverses the DOM tree (cf. 3.5(b)) from its root ($E_1 : ⟨div⟩$) in a bottom-up manner. When our general algorithm reaches the node $E_2$, it finds that $E_2$’s children $E_3$ and $E_4$ can be folded because they have the same tag string (“li”). Our general algorithm thus folds them and adds an iteration node “(for1)” as the parent of the node “⟨li⟩”. Also, our general algorithm extracts its corresponding environment whose variable name is also “for1”. For the node $E_5$, our general algorithm folds and aligns their children and the result is shown in Figure 3.5(c). We denote the nodes that are splitted into the same chunks with colors (orange and green) in Figure 3.5(b) and its corresponding layout trees generated in Figure 3.5(c). In Figure 3.5(c), the data records on the right side are the corresponding environment. The variable names “for1” and “for2” refers to the iteration nodes in the layout tree. “R1” and “R2” refer to the reference nodes and they replace the text nodes (“R1” replaces “T1” and “T2”, “R2” replaces “T3”, “T4” and “T5”) in the original DOM tree (cf. Figure 3.5(b)), because the text nodes’ values are different. Then, our general algorithm goes to the node “$E_1$”. It finds that $E_1$’s two children can be splitted into the same chunk (the region in red in Figure 3.5(c)). Our general algorithm folds them and aligns the different between $E_2$ and $E_5$. The result is shown in Figure 3.5(d). The right part of the Figure 3.5(d) is the final environment. In the final environment, “for1” consists of two objects (two “for2”). The first “for2” contains the reference variable “R1”, which comes from the green part in Figure 3.5(c). The second “for2” contains the reference variable “R2”, which comes from the orange part in Figure 3.5(d). The “for1” is before “for2” to maintain the order among siblings (two “⟨ul⟩” nodes).

Listing 3.1: A template program

```xml
<div>
  for for2 in for1:
</div>
```
Figure 3.5: An example of the separation algorithm
Next, we explain why our general algorithm works in a lossless manner by reconstructing the source code (Listing 3.2) based on the generated template code and its corresponding environment (cf. Figure 3.5(d)). Intuitively, the final layout tree (cf. Figure 3.5(d)) can be translated into template code (cf. Listing 3.1). “for1”, “for2” and “R1” are variable names. The template code has a nested loop (“for1” and “for2”). For every “R1” in “for2”, the template code has a loop to wrap every “R1” in “for2” with a tag “li”. Besides, for a list of “⟨li⟩R1⟨/li⟩”, the template code wraps them with a tag “ul”. Finally, a node “div” is added as the root and the source code (cf. Listing 3.2) of the DOM tree is reconstructed. Therefore, our general algorithm works in a lossless manner which means the source code of a web page can be reconstructed by executing the generated template code on the corresponding environment.
As given above, our general algorithm consists of three procedures: splitting, folding and alignment. The splitting algorithm works independently to split siblings into chunks to maximize the MI score of layout trees which are generated by folding and aligning siblings in the same chunk. In following sections, we utilize the math symbols shown in Table 3.2 to describe our proposed algorithms.

Table 3.2: Frequently Used Symbols

<table>
<thead>
<tr>
<th>Math Symbol</th>
<th>Conception</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>a DOM tree</td>
</tr>
<tr>
<td>BL</td>
<td>a boundary list</td>
</tr>
<tr>
<td>bl</td>
<td>a boundary</td>
</tr>
<tr>
<td>S</td>
<td>a list of siblings</td>
</tr>
<tr>
<td>S'</td>
<td>the layout trees after processing S'</td>
</tr>
<tr>
<td>S_p</td>
<td>the layout tree after processing S_p</td>
</tr>
<tr>
<td>i, j, p, q</td>
<td>index</td>
</tr>
<tr>
<td>S_p</td>
<td>a chunk of siblings</td>
</tr>
<tr>
<td>S_p[i]</td>
<td>the i\text{th} sibling of S_p[i]</td>
</tr>
<tr>
<td>s</td>
<td>a sibling</td>
</tr>
<tr>
<td>s_i</td>
<td>a sibling whose index is i</td>
</tr>
<tr>
<td>S_p[i : j]</td>
<td>i\text{th} to j\text{th} siblings in S_p</td>
</tr>
<tr>
<td>T[i]</td>
<td>the i\text{th} node of T</td>
</tr>
<tr>
<td>MI</td>
<td>the MI score</td>
</tr>
<tr>
<td>MI(S)</td>
<td>the MI score of S</td>
</tr>
<tr>
<td>Similarity(s_i, s_{i+1})</td>
<td>the similarity score between s_i and s_{i+1}</td>
</tr>
<tr>
<td>max_gen</td>
<td>the maximum number of iterations</td>
</tr>
<tr>
<td>max_wait</td>
<td>the maximum number of waiting iterations</td>
</tr>
</tbody>
</table>

In particular, we distinguish the meanings of some similar symbols and give them as follows:

- $MI(S_p)$ stands for the MI score of $S_p$, and $MI(S'_p)$ stands for the MI score of the layout tree which is generated by folding and align siblings in $S_p$.

- $MI(s_i, s_{i+1})$ stands for the MI score of $s_i$ and $s_{i+1}$, and $MI([s_i, s_{i+1}]')$ stands for the MI score of the layout tree by folding and aligning $s_i$ and $s_{i+1}$.

- $MI(S'_p, s_q)$ stands for the MI score of $S'_p$ and $s_q$, and $MI([S'_p, s_q]'')$ stands for the MI score of the layout tree by folding and aligning $S'_p$ and $s_q$. 
• $S'$ may contain more than one layout trees, and $S'_p$ contains only one layout tree.

For the maths symbols that are not explained in the above list, we will describe them in following sections.

### 3.4.2 Splitting Algorithms

The splitting algorithm aims to split a list of siblings into chunks to maximize the MI score of the layout trees after processing. For two continuous siblings $s_i$ and $s_{i+1}$, we consider following cases to split sibling:

- If $s_i$ and $s_{i+1}$ have different node types, the splitting algorithm splits $s_i$ and $s_{i+1}$ into two chunks.
- If $s_i$ and $s_{i+1}$ are text nodes, $s_i$ and $s_{i+1}$ are in the same chunk.
- If $s_i$ and $s_{i+1}$ are element nodes and either tag strings or attribute names are not the same, the splitting algorithm splits $s_i$ and $s_{i+1}$ into two chunks.
- If $s_i$ and $s_{i+1}$ are element nodes and both tag strings and attribute names are the same, we leave them for further processing.

It is worth emphasizing that the input siblings of the splitting algorithm are element nodes or text nodes. The reference node and instruction nodes are not included in the siblings because the siblings are layout trees that are returned from the general algorithm (cf. Line 7 in Algorithm 4). Therefore, if both $s_i$ and $s_{i+1}$ are text nodes, we split them into one chunk. If $s_i$ and $s_{i+1}$ are element nodes, we compare the tag strings and attribute names. The $s_i$ and $s_{i+1}$ could be in the same chunk only when the tag string and attribute names are the same. In any other cases, $s_i$ and $s_{i+1}$ should be split into two chunks. However, even when the tag strings and attribute names of $s_i$ and $s_{i+1}$ are the same, $s_i$ and $s_{i+1}$ may be split into two different chunks because their children have many differences and the reduction of SLOC, operators, and operands cannot overcome the increase of Cyclomatic Complexity brought by condition nodes. The reason is the alignment algorithm (detailed in Section 3.4.4) adds condition nodes ("⟨li⟩") to capture the differences among
Algorithm 5: Splitting(S)

Data: A list of siblings $S = s_1, s_2, \ldots, s_n$

1. \textbf{if} $S$.length equals to 1 \textbf{then}
2. \hspace{1em} \textbf{return} $S$;
3. // Initialization
4. \hspace{1em} $chunks = []$;
5. \hspace{1em} $tmp = [s_1]$;
6. \textbf{for} $i = 2 \rightarrow n$ \textbf{do}
7. \hspace{2em} \textbf{flag} = False;
8. \hspace{2em} \textbf{if} $tmp[-1]$ and $s_i$ are text nodes \textbf{then}
9. \hspace{3em} $tmp = tmp \cup s_i$;
10. \hspace{3em} \textbf{flag} = True;
11. \hspace{2em} \textbf{else if} $tmp[-1]$ and $s_i$ are element nodes \textbf{then}
12. \hspace{3em} \textbf{if} $tmp[-1].tag$ equals to $s_i.tag$ and $tmp[-1].attr_names$ equals to $s_i.attr_names$ \textbf{then}
13. \hspace{4em} $tmp = tmp \cup s_i$;
14. \hspace{4em} \textbf{flag} = True;
15. // split siblings
16. \hspace{2em} \textbf{if} \textbf{flag} is False \textbf{then}
17. \hspace{3em} $chunks = chunks \cup tmp$;
18. \hspace{3em} $tmp = [s_i]$;
19. // add $tmp$ to $chunks$ because $tmp$ is not null
20. \hspace{1em} add $tmp$ to $chunks$;
21. // Further split every chunk into small chunks
22. \textbf{for} $chunk$ in $chunks$ \textbf{do}
23. \hspace{2em} replace chunk with FurtherSplitting(chunk);

The general splitting algorithm begins with checking the number of siblings (Line 1). If $S$ has only one sibling, it wraps $S$ and returns the list of layout trees ($|S|$, $S$ is the layout tree, and there is only one layout tree in the list). Otherwise, the general splitting algorithm processes the siblings ($S$) pairwise to split adjacent siblings. The general splitting algorithm initializes a variable $chunks$ to store a list of chunks (Line 4) and a variable $tmp$ to store the currently processing chunk (Line 5). The general splitting algorithm puts the first sibling $s_1$ into $tmp$ to enable pairwise comparison by comparing the last element ($tmp[-1]$) in $tmp$ with $s_i$. From Line 6, the general splitting algorithm starts pairwise processing and utilizes a variable $\text{flag}$ (Line 7) to instruct whether $s_i$ should be in the
Currently processing chunk (tmp). If the value of flag is False, $s_i$ should not be inserted into tmp. Otherwise, $s_i$ should be inserted into tmp. If $s_i$ should be inserted into tmp (Line 8 and Line 12), the general splitting algorithm inserts the $s_i$ to the end of tmp and switch flag to True. Otherwise, the general splitting algorithm inserts tmp into chunks (Line 16) while cleans and initializes the tmp with $s_i$. After the loop ends, there should be some siblings in tmp. The general splitting algorithm thus inserts tmp into chunks (Line 19). Every chunk of chunks should be further split into small chunks if the splitting can achieve bigger maintainability index than none splitting.

We model the problem of splitting siblings into chunks as a problem of finding boundaries to split the siblings. We represent the boundary of $n$ siblings with a list of 0 or 1 whose length is $n - 1$, where 1 stands for a boundary (i.e., in different chunks), while 0 stands for none boundary (i.e., in the same chunk). The problem of finding the boundaries to separate siblings into chunks is defined as follows.

**Definition 34.** Finding Chunk Boundaries Given a list of siblings with the same tag string $S = [s_1, s_2, \ldots, s_n]$ and the maintainability index function, find a boundary list $BL = [bl_1, bl_2, \ldots, bl_{n-1}]$ that maximizes the MI score of the layout trees $S'$ that is generated by $S$ and $B$.

![Figure 3.6: A splitting example](image)

To help understand the significance of FindingChunkBoundaries, we give a running
example in the Figure 3.6. \( A_1, A_2, B_1, B_2 \) and \( B_3 \) are element nodes with the same tag string and attribute names. However, they are generated by different template code \( A \) and \( B \), which means that they have many differences. \( A_1 \) and \( A_2 \) are generated by \( A \), and \( B_1, B_2 \) and \( B_3 \) are generated by \( B \). The general splitting algorithm denote the layout tree from folding \( A_1 \) and \( A_2 \) by \( A' \), and the layout tree from folding \( B_1, B_2 \) and \( B_3 \) by \( B' \). The general splitting algorithm also denote the layout tree from folding \( A_1, A_2, B_1, B_2 \\) and \( B_3 \) by \( ALL \). If \( MI(ALL) < MI(A', B') \), the general splitting algorithm need to split \( A_1, A_2, B_1, B_2 \) and \( B_3 \) into two chunks ([\( A_1, A_2 \)] and [\( B_1, B_2, B_3 \)] respectively) to maximize the MI score of layout trees.

**Greedy Splitting Algorithm**

We first propose a greedy algorithm to solve the \textit{FindingChunkBoundaries} problem. The main idea is to keep the increase of MI score of layout trees after adding new siblings and insert a boundary when the MI score drops. Algorithm 6 summarizes the greedy splitting algorithm. \( S_p \) stands for a chunk of siblings whose tag strings and attribute names are the same. The greedy algorithm initializes a variable \( tmp \) to store currently processing chunk. Beginning with the second siblings in \( S_p \), the greedy splitting algorithm attempts to split the sibling \( s_{pi} \) into current processing chunk (the variable \( tmp \) in Line 5) and compares the MI score of the layout trees before and after folding and aligning \( tmp \) with \( s_{pi} \), which are \( MI([tmp, s_{pi}]') \) and \( MI(tmp, s_{pi}) \) respectively. If \( MI([tmp, s_{pi}]') \) is bigger than \( MI(tmp, s_{pi}) \), which means the MI score increases after inserting into the same chunk, we replace the chunk \( (tmp) \) with the chunk after inserting \( (tmp\_after) \). Otherwise, we create a new chunk which begins with \( s_{pi} \) and save the previous chunk \( (tmp) \) in the list of chunks \( (chunks) \). The splitting algorithm continues until all siblings in \( S_p \) are processed.

We use the Figure 3.6 to give a running example of the greedy splitting algorithm. The greedy splitting algorithm begins with a chunk \([A_1]\) (cf. \( tmp \) in Algorithm 6). The MI score of the layout tree which is constructed by folding \( A_1 \) and \( A_2 \) is bigger than the MI score of two layout trees \( (A_1 \ and \ A_2) \). We split \( A_1 \) and \( A_2 \) into one chunk and the \( tmp \) becomes \([A_1, A_2]\). Then, we try to insert \( B_1 \) into the \( tmp \) and compare the MI score of
Algorithm 6: Greedy Splitting Algorithm

Data: a subset of siblings $S_p$

1 // Initialization
2 $chunks = []$;
3 $tmp = [S_p[1]]$;
4 for $i = 2 \rightarrow q$ do
5     $tmp_after = tmp + s_p$;
6     if $MI(tmp_after) > MI(tmp, s_p)$ then
7         $tmp = tmp_after$;
8     else
9         $chunks = chunks \cup tmp$;
10        $tmp = [s_p]$;
11     end
12 $chunks = chunks \cup tmp$;
13 return $chunks$;

the layout tree which is from $[A_1, A_2, B_1]$ and the MI score of two layout trees ($[A_1, A_2]$ and $B_1$). As $B_1$ is generated from template code $B$ and $B$ is different from the template code $A$, the MI score of $[A_1, A_2, B_1]$ is smaller than the MI score of $[A_1, A_2]$ and $B_1$. We add $[A_1, A_2]$ into the list of chunks ($[[A_1, A_2]]$) and the chunk ($tmp$) becomes $[B_1]$. After processing all siblings, the chunk list is $[[A_1, A_2], [B_1, B_2, B_3]]$.

**TLBO splitting algorithm**

The greedy splitting algorithm cannot always get the best solution in our project. When $MI(S_p') > MI(S_p'[1 : i], S_p'[i + 1 : p])$ and $MI(S_p'[1 : i + 1]) < MI(S_p'[1 : i], S_p[i + 1])$, the greedy algorithm cannot get the best solution. We address this problem and propose a TLBO-based splitting algorithm to obtain globally optimal solution. We summarize the TLBO-based splitting algorithm via pseudo code in Algorithm 7. Line 1 initializes the population with $pop_size$ solutions (boundary lists). Every boundary list $BL_i = [bl_{i1}, bl_{i2}, \ldots, bl_{in}]$. Therefore, the population is a matrix $Population = [BL_1, BL_2, \ldots, BL_{pop_size}]$.
Algorithm 7: TLBO-based Splitting Algorithm

**Data:** a chunk of siblings $S_p$ whose length is $n + 1$,
the population size $pop\_size$,
the maximum number of generations $max\_gen$,
the maximum number of waiting $max\_wait$,
the learning rate $r$

1. randomly generate $pop\_size$ boundary lists as the population
   $Population = [BL_1, BL_2, \ldots, BL_{pop\_size}]$;
2. **while** $max\_gen! = 0$ and $max\_wait! = 0$ **do**
3.   find the best solution in $Population$ as the Teacher;
4.   calculate the mean $MEAN$ of solutions $Population$;
5.   $tmp\_best = Teacher$;
6.   // Teacher phase
7.   **for** every solution $BL_i$ in $Population$ **do**
8.       $BL_i^{new} = BL_i + r \times (Teacher - MEAN)$;
9.       **if** $MI(BL_i^{new}) > MI(BL_i)$ **then**
10.          $BL_i = BL_i^{new}$;
11.   // Student phase
12.   randomly select two students whose index is $p$ and $q$;
13. **for** every solution $BL_i$ in $Population$ **do**
14.       $BL_i^{new} = BL_i + r \times (BL_p - BL_q)$;
15.       **if** $MI(BL_i^{new}) > MI(BL_i)$ **then**
16.          $BL_i = BL_i^{new}$;
17.   **max\_gen = max\_gen - 1**;
18. **if** the best solution in $Population$ equals to $tmp\_best$ **then**
19.       $max\_wait = max\_wait - 1$;
20. **else**
21.       reset $max\_wait$;
22. **return** the best solution in $Population$;
3.4 Our Method

We calculate the mean of Population column-wise to get the $MEAN = [bl_1, bl_2, \ldots, bl_n]$. Line 3 selects the best solution from Population as the Teacher. For every solution $BL_i$, we get a new solution $BL_i^{new}$ by add the difference between Teacher and $MEAN$ multiplied by the learning rate $r$. Then, the TLBO-based splitting algorithm compares the new solution ($BL_i^{new}$) and the old solution ($BL_i$). The $BL_i$ is replaced by $BL_i^{new}$ if the new solution is better than the old solution ($BL_i^{new}$’s MI score is bigger than $BL_i$’s MI score). After the teacher phase, the TLBO-based splitting algorithm randomly selects two students whose index is $p$ and $q$ respectively. Lines 12 - 14 learn from the difference between $BL_p$ and $BL_q$ multiplied by the learning rate $r$. Lines 15 - 19 update the $max\_gen$ and $max\_wait$. $max\_gen$ is the maximum number of iterations, and the number of teaching and learning procedures cannot exceed this number. $max\_wait$ stands for the patience. If the best solution does not change after $max\_wait$ iterations, the iterations terminate before the maximum number of iterations is reached.

To help understand the TLBO-based splitting algorithm, we give a running example. For the siblings in Figure 3.6, the best solution is $[0, 1, 0, 0]$ which represents that splits $[A_1, A_2, B_1, B_2, B_3]$ into two chunks $[A_1, A_2]$ and $[B_1, B_2, B_3]$. To begin with, we randomly generate three solutions $[0, 1, 1, 1]$, $[1, 0, 1, 1]$ and $[0, 1, 0, 1]$. We then calculate the mean of our solution:

$$Population = \begin{bmatrix}
0 & 1 & 1 & 1 \\
1 & 0 & 1 & 1 \\
0 & 1 & 0 & 1
\end{bmatrix}$$

$$MEAN = \begin{bmatrix}
\frac{1}{3} & \frac{2}{3} & \frac{2}{3} & 1
\end{bmatrix}$$

We assume that $[0, 1, 0, 1]$ is the best solution:

$$Teacher = [0, 1, 0, 1]$$

Let the learning rate be 1. Every solution in $Population$ learns from the teacher:

$$[1, 0, 1, 1] + 1 \times ([0, 1, 0, 1] - \begin{bmatrix}
\frac{1}{3} & \frac{2}{3} & \frac{2}{3} & 1
\end{bmatrix}) = \begin{bmatrix}
\frac{2}{3} & \frac{1}{3} & 1 & 1
\end{bmatrix}$$
\[
\begin{align*}
[0, 1, 0, 1] &+ 1 \times (\left[0, 1, 0, 1 \right] - \left[\frac{1}{3}, \frac{2}{3}, \frac{2}{3}, \frac{1}{3} \right]) = \left[-\frac{1}{3}, \frac{4}{3}, \frac{2}{3}, \frac{1}{3} \right] \\
[0, 1, 1, 1] &+ 1 \times (\left[0, 1, 0, 1 \right] - \left[\frac{1}{3}, \frac{2}{3}, \frac{2}{3}, \frac{1}{3} \right]) = \left[-\frac{1}{3}, \frac{4}{3}, \frac{1}{3}, \frac{1}{3} \right]
\end{align*}
\]

After rounding to the nearest non-negative integers, these three new individuals become \([1, 0, 0, 1]\), \([0, 1, 0, 1]\) and \([0, 1, 0, 1]\). After replacing the old solution with better new solution, the population becomes \([1, 0, 0, 1]\), \([0, 1, 0, 1]\) and \([0, 1, 0, 1]\). Then, we randomly select two students which are \([1, 0, 0, 1]\) and \([0, 1, 0, 1]\). Then, we learn from these two students:

\[
\begin{align*}
[1, 0, 0, 1] &+ 1 \times (\left[0, 1, 0, 1 \right] - \left[1, 0, 0, 1 \right]) = \left[0, 1, 0, 1 \right] \\
[0, 1, 0, 1] &+ 1 \times (\left[0, 1, 0, 1 \right] - \left[1, 0, 0, 1 \right]) = \left[-1, 2, 0, 1 \right]
\end{align*}
\]

After rounding to the 0 and 1 and replacing old solution with better new solutions, the population becomes \([0, 1, 0, 1]\). \([0, 1, 0, 1]\) is the best solution calculated by the TLBO-based splitting algorithm. However, it is different from the best solution \([0, 1, 0, 0]\). Conventional population-based optimization algorithms are not able to identify the contributions of every element in the solution. We modify the algorithm and propose an algorithm named dual-TLBO in the next section.
Dual Teaching and Learning Based Optimization Algorithm

We design a population-based algorithm which is named dual teaching and learning based optimization (dual-TLBO) to solve Finding Chunk Boundaries problem. Figure 3.7 shows the key steps of our dual-TLBO. First, the algorithm initializes the population and a termination criterion. The termination criterion is the maximum number of loops. Each individual in the population is a list of 0’s and 1’s, which represents a solution \( B \) of the “Finding Chunk Boundaries” problem. It is initialized randomly. The mean of all individuals in the population is computed. Then a teacher phase starts. In this phase, we get a new for every individual \( X_{old} \), we create a new individual \( X_{new} \) based on the teacher and the mean value, which is denoted by

\[
X_{new} = X_{old} + r \times (Teacher - Mean),
\]

where \( r \) is a system parameter between 0 and 1 that represents the learning rate. The computed individual \( X_{new} \) may contain non-integer numbers or negative numbers. We round the
number to its nearest non-negative integer. If $X_{\text{new}}$ represents a chunking that yields a higher MI score, we replace $X_{\text{old}}$ by $X_{\text{new}}$. Otherwise, $X_{\text{old}}$ is kept and $X_{\text{new}}$ is discarded. Then, it comes to the student phase. We randomly select two individuals $X_i$ and $X_j$ and for every individual $X_{\text{old}}$, we create a new individual $X_{\text{new}} = X_{\text{old}} + r(X_i - X_j)$. Similar to the teacher phase, we compare $X_{\text{old}}$ with $X_{\text{new}}$ and keep the one producing a higher MI score. We add it a dual step to the algorithm, where we run another teacher phase and another student phase but learn from the worst (rather than the best) individual of the original population. After learning from the worst and the best individuals, we can compare the new best individual and the new worst individual element-wise. If the element in the same position is the same, we consider it as a not well-learned element because the contribution to improving maintainability index of the element is not clear. For these positions, we mutate their values and keep the new best solution for the next iteration. If the termination criterion is met, the algorithm returns the best solution, otherwise found so far. Algorithm 8 summarizes the dual-TLBO algorithm.

To help understand the dual-TLBO algorithm, we give a running example next. We assume 5 siblings. The workflow is shown in Figure 3.8. First, we randomly generate three individuals: $[0, 1, 1, 1], [1, 0, 1, 1]$ and $[0, 1, 0, 1]$. We assume that $[0, 1, 0, 1]$ is the current best individual. The mean of the population is $[1/3, 2/3, 2/3, 1]$ and every individual can generate three new individuals which are $[2/3, 1/3, 2/3, 1], [−1/3, 4/3, −2/3, 1]$ and $[−1/3, 4/3, 1/3, 1]$. After rounding to the nearest non-negative integers, these three new individuals become $[1, 0, 0, 1], [0, 1, 0, 1]$ and $[0, 1, 0, 1]$. In this step, we only replace the
Algorithm 8: Dual-TLBO

Data: A Boundary List $B$, 
the size of population $pop_size$, 
the learning rate $r$

1 // Initialization
2 Randomly generate $pop_size$ solutions as the initial population $P$;
3 $m = \text{Mean}(P);$ 
4 $best\_solution = \text{FindBestSolution}(P);$ 
5 $worst\_solution = \text{FindWorstSolution}(P);$ 
6 $best\_population = P;$ 
7 $worst\_population = P;$ 
8 while termination criterion is not meet do
9  // Teacher Phase
10  for $i = 1 \rightarrow pop\_size$ do 
11     $p = best\_population[i] + r \times (best\_solution[i] - m);$ 
12     if $MI(p) > MI(best\_population[i])$ then 
13         $best\_population[i] = p;$ 
14    for $i = 1 \rightarrow pop\_size$ do 
15     $p = worst\_population[i] + r \times (worst\_solution[i] - m);$ 
16     if $MI(p) < MI(worst\_population[i])$ then 
17         $worst\_population[i] = p;$ 
18  // Student Phase
19  randomly select two solutions (the index is x and y and 
20     $MI(best\_population[x]) > MI(best\_population[y])$ from best\_population;
21  for $i = 1 \rightarrow pop\_size$ do 
22     $p = best\_population[i] + r \times (best\_solution[x] - best\_solution[y]);$
23     if $MI(p) > MI(best\_population[i])$ then 
24         $best\_population[i] = p;$
25  randomly select two solutions (the index is x and y and 
26     $MI(worst\_population[x]) < MI(worst\_population[y])$ from worst\_population;
27  for $i = 1 \rightarrow pop\_size$ do 
28     $p = worst\_population[i] + r \times (worst\_solution[x] - worst\_solution[y]);$
29     if $MI(p) < MI(worst\_population[i])$ then 
30         $worst\_population[i] = p;$ 
31  // Mutation
32  $best\_solution = \text{FindBestSolution}(P);$ 
33  $worst\_solution = \text{FindWorstSolution}(P);$ 
34  Compare $best\_solution$ and $worst\_solution$ pairwise and mutate values in 
35  $best\_solution$, if the solution after mutation is better than $best\_solution$, 
36  replace $best\_solution$ with the new solution;
37 return $best\_solution;$
old individuals that are worse than the new individuals. After learning from the teacher, the population becomes \([1,0,0,1], [0,1,0,1]\) and \([0,1,0,1]\). Then, we randomly select two students from the population and the students are \([1,0,0,1], [0,1,0,1]\). Every student learns from these two students, and the population becomes three \([1,0,0,1]\). Then, we use the original population: \([0,1,1,1], [1,0,1,1]\) and \([0,1,0,1]\) to learn from the worst individual. In this step, we aim to minimize the MI score of individuals. We assume that \([1,0,1,1]\) is the current worst individual. After learning from them, the bad population becomes three \([1,0,1,1]\). Compare the best individual \([0,1,0,1]\) and the worst individual \([1,0,1,1]\). We find that the last element in these two individuals is the same. This is a non-well learned element. We mutate it and update the best individual to be \([0,1,0,0]\).

### 3.4.3 Folding Algorithm

We propose a folding algorithm to fold siblings \((S_p)\) in one chunk into a layout tree with a “for” node (denoted by \(S'_p = for(S_p)\)). The folding algorithm detects the common parts among siblings, and the alignment algorithm locates the differences among siblings. The folding algorithm and alignment algorithm work together to construct the layout tree \((S'_p)\) and its corresponding environment \((E'_p)\). The chunk of siblings \((S_p)\) can be reconstructed from the layout tree \((S'_p)\) and its corresponding environment \((E'_p)\). \(S'_p\) has a single child which can be an element node, or a text node, or a reference node referred to text nodes.

To fold a chunk of siblings, we propose an incrementally increase algorithm. Our folding algorithm let the second sibling fold first sibling \(s_{p1}\) and continue until all siblings are folded. We summarize the folding algorithm by pseudo code in Algorithm 9. We consider the \(S_p\) as a queue (Line 1) and dequeue siblings from \(S_p\) until all siblings are processed. We fold the siblings one by one and finally add a “for” root as the parent of the layout tree.

The difference between folding algorithms and alignment algorithms (detail in next section, Section 3.4.4) is that folding algorithms aim to capture common parts, and alignment algorithms capture the different parts. When folding siblings, we add “for” nodes to capture the common parts of siblings. The differences among siblings are captured
Algorithm 9: Folding Algorithm

Data: a chunk of siblings $S_p$

1. $S^\text{for}_p = \text{dequeue}(S_p)$;
2. while $S_p$ is not empty do
   3. $S^\text{for}_p = \text{fold}(S^\text{for}_p, \text{dequeue}(S_p))$;
   4. add “for” node as the parent of $\text{for}(S_p)$;
5. return $\text{for}(S_p)$;

3.4.4 Alignment Algorithm

The alignment algorithms (alignment procedure) captures the differences among siblings by adding reference nodes and condition nodes and establishing corresponding environments. We capture three types of differences which are shown as follows:

- The attributes of element nodes
- Text nodes
- Additional nodes

If the attribute names of element nodes are different, the alignment procedure considers them as additional nodes and adds a condition node as the parent of each element node. If the attribute names are the same while the attribute values are different (cf. “href” in Figure 3.1(a)), the alignment procedure utilizes reference nodes to capture the differences and refers them to the corresponding environment. Reference nodes which refer to attribute values are processed with element nodes (“r1” in Figure 3.1(b)). In contrast, the reference nodes which replace text nodes (“r2”) is processed like an element node. For reference nodes which refers to text node, we treat them as an element whose tag string
is “text_node” and “text_node” is unique. Additional branches are element nodes, text nodes or iteration nodes. We add “if” node as the parent of additional branches.

Alignment Operations

Given two siblings $s_i$ and $s_{i+1}$ and their corresponding environments $E_i$ and $E_{i+1}$, the alignment algorithm aligns them to capture differences and cooperates with the folding algorithm to generate one layout tree ($s'$) and also their environments ($E'$).

**Text nodes alignment** Both $s_i$ and $s_{i+1}$ are text nodes. If the text value of $s_i$ and $s_{i+1}$ are the same, we treat them as string and they are still text nodes after processing. Otherwise, we add a reference node to replace $s_i$ and $s_{i+1}$ in the layout tree. In particular, we add a variable in the environment to refer to the siblings. We show an example of text node alignment in Figure 3.9. Every figure shows the original tree and the layout tree after alignment. When the text nodes have the same value (“Apple”, cf. Figure 3.9(a)), the node after alignment is still a text node whose value is “Apple”. However, when the text nodes have different values (“Apple” and “Banana”, cf. Figure 3.9(b)), we add a reference node (“r1”) to replace them and add variables (“r1: Apple, r1: Banana”) to the environment to record the difference.

**Text node and reference node alignment** We assume that “$s_i$” is a reference node and “$s_{i+1}$” is a text node. We use a reference node in the layout tree and add the value of text node to the environment of $s_i$ ($E_i$). We given an example in Figure 3.10 “r1” is a reference node and its corresponding environment is “r1: Apple, r1: Banana”. We aim to align the text node whose value is “Candy” with “r1”. We discard the text node from the layout tree and add its value to the environment. The results are shown in the right side of Figure 3.10. During alignment, the order of siblings are maintained (e.g. “r1: Candy” is added after “r1: Apple, r1: Banana”).

Listing 3.3: Tag strings are different
3.4 Our Method

(a) Text nodes have the same value

<table>
<thead>
<tr>
<th>Layout Tree</th>
<th>Before Alignment</th>
<th>After Alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Apple</td>
<td>Apple</td>
<td>Apple</td>
</tr>
<tr>
<td>Environment</td>
<td>{}</td>
<td>{}</td>
</tr>
</tbody>
</table>

(b) Text nodes have different values

<table>
<thead>
<tr>
<th>Layout Tree</th>
<th>Before Alignment</th>
<th>After Alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Apple</td>
<td>Banana</td>
<td>{r1}</td>
</tr>
<tr>
<td>Environment</td>
<td>{}</td>
<td>[ {r1: Apple}, {r1: Banana} ]</td>
</tr>
</tbody>
</table>

Figure 3.9: Text nodes alignment
Element nodes alignment When both \( s_i \) and \( s_{i+1} \) are element nodes, we align them only when the tag strings and attribute names are the same. For example, in Listing 3.3, tag string “h3” and “div” are different, we thus cannot align them. In Listing 3.4, although the tag strings are the same (both tag strings are “div”), their attributes are not the same. The first “div” node’s attribute names are “href” (stands for a link) and “class” (refers to a stylish file). The second node’s attribute names are “index” and “href”. We cannot align the cases given above and keep it as they are in the layout tree. For Listing 3.5, the tag
### 3.4 Our Method

#### (a) Attribute values are the same

<table>
<thead>
<tr>
<th>Layout Tree</th>
<th>Before Alignment</th>
<th>After Alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Tree Diagram" /></td>
<td><img src="image2" alt="Tree Diagram" /></td>
<td><img src="image3" alt="Tree Diagram" /></td>
</tr>
</tbody>
</table>

| Environment | {} | {} |

#### (b) Attribute values are not the same

<table>
<thead>
<tr>
<th>Layout Tree</th>
<th>Before Alignment</th>
<th>After Alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image4" alt="Tree Diagram" /></td>
<td><img src="image5" alt="Tree Diagram" /></td>
<td><img src="image6" alt="Tree Diagram" /></td>
</tr>
</tbody>
</table>

| Environment | {} | [ {r1: product}, {r1: price} ] |

#### (c) Reference nodes in attribute values

<table>
<thead>
<tr>
<th>Layout Tree</th>
<th>Before Alignment</th>
<th>After Alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image7" alt="Tree Diagram" /></td>
<td><img src="image8" alt="Tree Diagram" /></td>
<td><img src="image9" alt="Tree Diagram" /></td>
</tr>
</tbody>
</table>

| Environment | {} | [ {r1: product}, {r1: price}, {r1: condition} ] |

Figure 3.11: Element nodes alignment
strings (“div”) and attribute names (“index” and “href”) are the same. Therefore, they can be aligned.

If the attribute values of element nodes are the same, we add the element node to the layout tree. As shown in Figure 3.11(a), the two “li” nodes have the same attribute values (class = ‘product’). Therefore, we add the element node to the layout tree and align their children recursively. If the attribute values of element nodes are different, we add a reference node to represent them in the layout tree and add the corresponding variable to the environment. In Figure 3.11(b), the attribute values are not the same (“product” and “price”). We add a reference node “r1” to the layout tree and variables to the environment (“r1: product, r1: price”) as well. When there exist reference nodes in attribute names (e.g. “r1” in Figure 3.11(c)), we add the attribute value to the reference node. We perform the same procedure when there are reference nodes in both siblings. Also, the children of $s_i$ and $s_{i+1}$ are aligned recursively.

**Null alignment** If a node ($s_i$) is considered as an additional branch, the alignment algorithm needs to instruct that it is an additional branch in both the layout tree and its environment. As it is like aligning $s_i$ with null, we name this procedure as a null alignment. If $s_i$ is an element node or a text node, the alignment algorithm adds a “if” node as the root of the node to construct the layout. Also, the alignment algorithm wraps the corresponding environment $E_i$ of $s_i$ to construct the environment of the layout tree. For example, in Figure 3.1(b), the “span” node is an additional branch. The alignment algorithm adds a “if1” node to instruct that the “span” node is not shared by all the siblings generated by the same template code. Besides, the alignment algorithm adds “if1” to the environment. As the environment of the branch is empty, the “if1” maps to empty in the environment of the layout tree after alignment.

If the $s_i$ is a condition node, the alignment algorithm returns the $s_i$ as the layout tree after alignment. If the $s_i$ is an iteration node, the alignment algorithm wraps $s_i$ with a condition node and returns the “if” layout tree. It is worth noticing that an iteration node
Figure 3.12: An example of null alignment of iteration nodes
can be a first-level child of a condition node, while a condition node cannot be a first-level child of an iteration node. We use the example in Figure 3.1 to explain. When the “span” node is an additional branch, the alignment algorithm wraps the “span” with an iteration node (cf. “if1” in Figure 3.1(b)). The iteration node (“if1”) must be a child of an element node (the “li” node), and the element node (the “li” node) can be a child of an iteration node (the “for1” node). Therefore, a condition node cannot be a child of an iteration node. In Figure 3.12, the “for1” node is an additional branch because it is not in the other “div” node. The alignment algorithm wraps the “for1” node with a “if1” node. Also, the “for1” variable in the environment is also wrapped by a “if1” variable.

**Element node and instruction node alignment** If $s_i$ is an instruction node and $s_{i+1}$ is an element node, aligning $s_i$ and $s_{i+1}$ is an element-instruction alignment. If $s_i$’s first-level child and $s_{i+1}$ are (1) text nodes, or (2) element nodes with the same attribute names and tag strings, or (3) reference nodes, $s_i$ and $s_{i+1}$ can be aligned. The alignment algorithm first creates an instruction node in the layout tree and then aligns $s_i$’s first-level child with $s_{i+1}$ via the alignment operations given above. While alignment, the order of environments should be maintained.

**Instruction nodes alignment** If both $s_i$ and $s_{i+1}$ are instruction nodes, the alignment algorithm aligns the first child of $s_i$ and $s_{i+1}$ which is not an instruction node. Also, their first children who are not instruction nodes should be the same type, which means both of them are (1) text nodes, or (2) element nodes with the same tag string and attribute names, or (3) reference nodes.

**Global Alignment Algorithm**

Our alignment algorithm aligns siblings in the same chunk and captures the differences among the siblings. If sibling text nodes have different text strings, or sibling element nodes have different attribute values, the alignment algorithm adds a reference node, which refers to variables in the environment, to record the difference. If the structure of sibling nodes is different, we add condition nodes as parents of branches.
For every two siblings (which have been converted to layout trees already since our algorithm works in a bottom-up manner) in the same chunk, our algorithm aligns their children globally and recursively. Let the two layout trees be $A$ and $B$. The reference nodes, which refer to attributes, are aligned as the attributes of element nodes. Besides, the reference nodes, which replace text nodes, are kept. Therefore, reference nodes which refer to attributes are not included in $A$ or $B$. The score of aligning $A$ and $B$ is the difference between the MI score of after and before aligning $A$ and $B$. We consider alignment operations which are shown as below:

- Aligning a layout tree with null means adding a condition as the root of the layout tree. If the root of the layout tree is already a condition node, the layout tree does not change. We denote the layout trees of align $A$ and $B$ with null by $A_c$ and $B_c$ respectively.

- If at least one of $A$ and $B$ has an instruction node as the root, the alignment begins from their child, which is not an instruction node and the instruction node is kept as the root of the layout tree after aligning.

- If the roots of $A$ and $B$ are element nodes, and their attribute names and tag string are the same, then they become one element node and their children are aligned recursively.

- If $A$ and $B$ are reference nodes, which refer to text nodes, they become one reference node.

Let the number of children of $A$ and $B$ be $m$ and $n$ and the $i$th child of $A$ and the $j$th child of $B$ be $A[i]$ and $B[j]$. The algorithm fills a matrix $ALIGN$ of size $(m + 1) \times (n + 1)$. $ALIGN[i, j]$ is the maximum value of (1)$ALIGN[i - 1, j - 1] + \text{score}(A[i], B[j])$, (2)$ALIGN[i - 1, j] + \text{score}(A[i])$ and (3)$ALIGN[i, j - 1] + \text{score}(B[j])$. The alignment algorithm fills $ALIGN$ from $ALIGN[0, 0]$ to $ALIGN[m + 1, n + 1]$ row by row. When the filling is done, $ALIGN[m + 1, n + 1]$ stores the maximum score of an alignment. We trace back from $ALIGN[m + 1, n + 1]$ to $ALIGN[0, 0]$ to obtain the alignment.
be the root of the new layout tree. During tracing back, we add children to \( T_a \) to generate an aligned layout tree. If \( \text{ALIGN}[i,j] = \text{ALIGN}[i-1,j+1] + \text{score}(A[i],B[j]) \), \( A[i] \) and \( B[j] \) are matched, we align them and add the generated layout tree to \( T_a \). If \( \text{ALIGN}[i,j] = A[i-1, j] + \text{score}(A[i]) \), we add a condition node as the parent of \( A[i] \) to generate \( A_c[i] \) and add \( A_c[i] \) to \( T_c \). If \( \text{ALIGN}[i,j] = A[i,j-1] + \text{score}(B[j]) \), we add a condition node as the parent of \( B[j] \), which is \( B_c[j] \), and add \( B_c[j] \) to \( T_c \). Besides, the environments are also aligned to adapt to the changes of the layout tree. The alignment algorithm returns \( T_c \).

To help understand our alignment algorithm, we give a running example. In Tables 3.3 and 3.4, where \([A_1, B_1, D_1, F_1]\) and \([A_2, C_2, D_2, E_2, F_2]\) are two lists of siblings. Table 2 shows the score of matching every branch with others from the other list. We fill the Table 3 from left to right in a top-down manner. Finally, we trace back from the last element of Table 3.4 which is \((F_2, F_1)\). The cells in orange show the matching path. The alignment result is \((E_1), (A_1, A_2), (C_1, B_2), (D_1, D_2), (F_1, F_2)\) and it increases the maintainability index by 12. Our alignment adds a condition node as the parent of \( E_2 \). Although \( \text{score}(C_1, B_2) \)
3.5 Experiments

In Section 3.5.1, we present the data sets and hardware used for the experiments and the parameters. We then show the results of our experiments in Section 3.5.2. We next analyze the effects of hyperparameters in Section 3.5.3. Last, we analyze the time efficiency to suggest the best parameters for different testbeds in Section 3.5.4.

3.5.1 Settings

Data Sets

To evaluate the empirical performance of our algorithms, we use four real data sets: SYNTHIA-DATASET [73], TBDW [108], UW-CAN-DATASET [41] and 4Universities (The 4 Universities Data Set) [28, 29, 36].

SYNTHIA-DATASET is a data set from the SYNTHIA paper [73]. SYNTHIA-DATASET consists of 200 web pages from 8 categories. It covers movies, vehicles, real-estates, forums, sports, jobs, e-commerce, and photography websites. There are 40 websites, and each website provides five different pages.

TBDW [108] is a data set for information extraction. We use version 1.02 which is published on 2004/06/09 and contains 253 web pages from 51 websites.

UW-CAN [41] is a data set of web pages for web data mining purposes. It contains 314 web pages from various web pages from the website of the University of Waterloo and other Canadian websites. It covers 10 categories which are news on the black bear attack, campus network, Canada transportation roads, career services, cooperations, health services, river fishing, river rafting, snowboarding skiing and winter Canada.
4Universities is a data set that contains web pages from computer science departments of various universities including Cornell University (867 web pages), Texas University (827 web pages), Washington University (1205 web pages), Wisconsin University (1263 web pages) and other universities (4120 web pages). The 8282 pages can be classified into seven categories: student (1641 web pages), faculty (1124 web pages), staff (137 web pages), department (182 web pages), course (930 web pages), project (504 web pages) and other (3764 web pages). The “other” category contains web pages that are not an instance of previous six categories, such as a list of publications web page or a research interest web page. We use the web pages of faculty and students respectively. We denote the faculty dataset as 4UNI-Faculty and the student dataset as 4UNI-Student. 4UNI-Faculty consists of 1641 web pages, and 4UNI-Student consists of 1124 web pages.

Besides, SYNTHIA-DATASET, 4Universities are grouped by websites. Intuitively, a website usually has a series of templates. We can compare the performance of our methods on different templates which can help us perform error analysis. We carry out experiments on all datasets given above and further analyze the efficiency and effects of parameters on SYNTHIA-DATASET.

**Hardware and Running Environment**

We perform our experiments in a MacBook Pro laptop with an Intel Core i7 processor. The RAM is 16 GB. We write the code in Python. We use a parser named “html5lib” to parse the web pages and process them with a Python library named Beautiful Soup. Beautiful Soup is a Python library for extracting data from HTML and XML documents. It provides abundant functions to handle the DOM tree which is parsed by “html5lib” from a web page.

**Parameters**

There are four parameters in dual-TLBO: population size, learning rate, the maximum number of generations and patience number. The population size equals the number of
siblings of a chunk plus one. The initial population consists of a boundary list are filled with 1’s and a boundary list which are filled with 0’s. For the others, we first initialize a boundary list with all 0’s and mutate one of the element into 1. The position of elements mutated into 1 is unique among the initial population. The initialization of the population provides enough solutions for the mutation phase. As every solution learns from the teacher and representative students pairwise, dual-TLBO can learn all elements pairwise if all elements cover all possible values (0 and 1) in the initial population. We set the learning rate as 0.9. The maximum number of generations is 25, and the patience number is 10. During the generations, the patience number is set to 10 when the best solution changes after a generation. After every generation, the maximum number of generations and patience number subtract one until one of them becomes zero. Then, the generations terminate and we output the best solution.

3.5.2 Results

We evaluate the empirical performance of our algorithms on real data sets given in the last section and report the results in this section. We show that our algorithms can produce separations with higher MI scores than those by the baseline algorithms. Meanwhile, our algorithms also produce separations with small sizes, which are comparable to those produced by the SYNTIA algorithm that is designed to produce separations with small sizes. As the code of SYNTIA is not available, we do our best to implement the algorithm following the SYNTIA paper [73] for our experiments. The result of experiments on SYNTIA-DATASET is shown in Table 3.5.

In the table, $MI$ denotes the maintainability index score. $CR$ denotes the compression ratio, i.e., the output file size (both the layout tree and corresponding environment) over the original file size (the original HTML document), and time denotes the algorithm response time. We also report $CR$ because SYNTIA is designed to achieve a small $CR$ value. We report the performance of five algorithms, $SYNTIA$ is our implementation of the SYNTIA algorithm [73]; $MAX - MI$ uses the same algorithmic steps of

\footnote{The compression ratio reported in the original SYNTIA paper is 62.8%.}
Table 3.5: SYNTHIA-DATASET

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>MI</th>
<th>CR</th>
<th>time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original page</td>
<td>-3.46 (0%)</td>
<td>100%</td>
<td>-</td>
</tr>
<tr>
<td>SYNTHIA</td>
<td>-3.06 (+0.8%)</td>
<td>63.34%</td>
<td>310</td>
</tr>
<tr>
<td>MAX-MI</td>
<td>5.83 (+17.8%)</td>
<td>66.30%</td>
<td>317</td>
</tr>
<tr>
<td>MAX-MI-LCS</td>
<td>6.97 (+20.0%)</td>
<td>66.31%</td>
<td>315</td>
</tr>
<tr>
<td>MI-Greedy</td>
<td>7.20 (+20.46%)</td>
<td>68.89%</td>
<td>370</td>
</tr>
<tr>
<td>MI-TLBO</td>
<td>7.55 (+21.13%)</td>
<td>66.43%</td>
<td>1466</td>
</tr>
<tr>
<td>MI-Dual-TLBO</td>
<td>9.38 (+24.5%)</td>
<td>67.27%</td>
<td>2031</td>
</tr>
</tbody>
</table>

SYNTHIA but uses the MI score to determine whether a subset of siblings should be folded; MAX – MI – LCS further adds the alignment algorithm described in Section 3.4.4. MI – Greedy uses the greedy algorithm (details in [23]) with the alignment algorithm; MI – TLBO replaces the greedy algorithm with traditional TLBO algorithm (detailed in [12]), while MI – Dual – TLBO is our proposed algorithm as described in Section 22.

We can see from the table that the proposed algorithm MI – Dual – TLBO outperforms all the baseline algorithms in terms of the MI score, while its compression ratio is very close to that of SYNTHIA which is designed to optimize the compression ratio. Compared with the MI score of original web pages, the SYNTHIA improves 0.8% and the MAX – MI improves 17.8%. Our greedy algorithm (MI-Greedy) achieves a MI score of 7.2 with an improvement of 20.46%. MI – TLBO outperforms the MI-Greedy with a MI score of 7.55. MI-Dual-TLBO achieves the state-of-the-art result with a MI score of 24.5. It dramatically improves the MI score from -3.46 to 9.38 with comparison of the original web pages. We notice that MI – Dual – TLBO is slower than SYNTHIA. We argue that web page separation is usually an offline task which can allow a slower algorithm. Further, if the algorithm response time is critical, then our adapted algorithm MAX – MI, MAX – MI – LCS and MI – Greedy can be used instead of SYNTHIA as they are as fast as SYNTHIA while obtaining larger MI scores.

We also test our algorithm on TBDW, the result is shown in Table 3.6. The MI – Dual – TLBO algorithm also outperforms all baseline algorithms on TBDW. The original web
3.5 Experiments

Table 3.6: TBDW

<table>
<thead>
<tr>
<th></th>
<th>MI</th>
<th>CR</th>
<th>time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original page</td>
<td>17.80</td>
<td>100%</td>
<td>-</td>
</tr>
<tr>
<td>SYNTHIA</td>
<td>23.02</td>
<td>74.19%</td>
<td>438</td>
</tr>
<tr>
<td>MAX-MI</td>
<td>26.72</td>
<td>75.58%</td>
<td>186</td>
</tr>
<tr>
<td>MAX-MI-LCS</td>
<td>27.36</td>
<td>74.67%</td>
<td>189</td>
</tr>
<tr>
<td>MI-Greedy</td>
<td>25.6</td>
<td>81.57%</td>
<td>211</td>
</tr>
<tr>
<td>MI-TLBO</td>
<td>27.45</td>
<td>74.48%</td>
<td>1488</td>
</tr>
<tr>
<td>MI-Dual-TLBO</td>
<td>28.48</td>
<td>75.62%</td>
<td>2066</td>
</tr>
</tbody>
</table>

Pages have a MI score of 17.8. The SYNTHIA can improve the MI score from 17.8 to 23.02 with a compression ratio of 74.19. MAX-MI archives a MI score of 26.72 with an improvement of 22.46%. MAX-MI-LCS achieves a MI score of 27.36. However, the MI-Greedy does not perform as well as MAX-MI and MAX-MI-LCS. It only achieves a MI score of 25.6. It proves that the MI-Greedy cannot always improves the MI score. MI-TLBO and MI-Dual-TLBO achieves 27.45 and 28.48 respectively. The MI-Dual-TLBO outperforms all other methods. Also, all these methods achieve a compression ratio of approximate 75%. The MI-Greedy only performs a compression ratio of 81.57%. Considering that the MI-Greedy achieves a MI score of 25.6, the MI-Greedy cannot split siblings into chunks effectively because the compression ratio is less than others.

Table 3.7: UW-CAN

<table>
<thead>
<tr>
<th></th>
<th>MI</th>
<th>CR</th>
<th>time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original page</td>
<td>42.45</td>
<td>100%</td>
<td>-</td>
</tr>
<tr>
<td>SYNTHIA</td>
<td>45.0</td>
<td>82.36%</td>
<td>127</td>
</tr>
<tr>
<td>MAX-MI</td>
<td>47.17</td>
<td>84.37%</td>
<td>147</td>
</tr>
<tr>
<td>MAX-MI-LCS</td>
<td>47.81</td>
<td>84.97%</td>
<td>125</td>
</tr>
<tr>
<td>MI-Greedy</td>
<td>47.10</td>
<td>85.79%</td>
<td>138</td>
</tr>
<tr>
<td>MI-TLBO</td>
<td>48.23</td>
<td>83.05%</td>
<td>1066</td>
</tr>
<tr>
<td>MI-Dual-TLBO</td>
<td>48.82</td>
<td>84.16%</td>
<td>1610</td>
</tr>
</tbody>
</table>

We also test our algorithms on UW-CAN, and the result is shown in Table 3.7. The original web page has a MI score of 42.45. SYNTHIA achieves a MI score of 45.0 with a compression ratio of 82.36%. MAX-MI increases the MI score to 47.17 with a decrease of
compression ratio (84.37%). MAX-MI-LCS achieves comparable MI score which is 47.81. Achieving the same trend with the experiment on TBDW, the MI-Greedy also does not outperform MAX-MI and MAX-MI-LCS with a MI score of 47.10. MI-TLBO and MI-Dual-TLBO achieves MI scores of 48.23 and 48.82. The MI-Dual-TLBO achieves the largest MI score among all methods and improves the MI score of original wbe pages with percentage of 11.1%.

Table 3.8: 4UNI-Faculty

<table>
<thead>
<tr>
<th></th>
<th>MI</th>
<th>CR</th>
<th>time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original page</td>
<td>63.57 (0%)</td>
<td>100%</td>
<td>-</td>
</tr>
<tr>
<td>SYNTHIA</td>
<td>66.98 (+4.3%)</td>
<td>93.53%</td>
<td>18</td>
</tr>
<tr>
<td>MAX-MI</td>
<td>67.47 (+4.9%)</td>
<td>93.97%</td>
<td>11</td>
</tr>
<tr>
<td>MAX-MI-LCS</td>
<td>67.90 (+5.5%)</td>
<td>93.98%</td>
<td>11</td>
</tr>
<tr>
<td>MI-Greedy</td>
<td>67.98 (+5.6%)</td>
<td>94.05%</td>
<td>13</td>
</tr>
<tr>
<td>MI-TLBO</td>
<td>68.33 (+6.1%)</td>
<td>93.69%</td>
<td>40</td>
</tr>
<tr>
<td>MI-Dual-TLBO</td>
<td>68.44 (+6.2%)</td>
<td>93.64%</td>
<td>388</td>
</tr>
</tbody>
</table>

Table 3.9: 4UNI-Student

<table>
<thead>
<tr>
<th></th>
<th>MI</th>
<th>CR</th>
<th>time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original page</td>
<td>64.64 (0%)</td>
<td>100%</td>
<td>-</td>
</tr>
<tr>
<td>SYNTHIA</td>
<td>66.43 (+2.2%)</td>
<td>92.17%</td>
<td>21</td>
</tr>
<tr>
<td>MAX-MI</td>
<td>67.08 (+3.1%)</td>
<td>94.59%</td>
<td>13</td>
</tr>
<tr>
<td>MAX-MI-LCS</td>
<td>67.52 (+3.6%)</td>
<td>94.23%</td>
<td>13</td>
</tr>
<tr>
<td>MI-Greedy</td>
<td>67.4 (+3.4%)</td>
<td>94.21%</td>
<td>14</td>
</tr>
<tr>
<td>MI-TLBO</td>
<td>67.9 (+4.1%)</td>
<td>93.71%</td>
<td>38</td>
</tr>
<tr>
<td>MI-Dual-TLBO</td>
<td>67.95 (+4.2%)</td>
<td>93.72%</td>
<td>141</td>
</tr>
</tbody>
</table>

The results of experiments on 4UNI-Faculty and 4UNI-Student are shown in Table 3.8 and Table 3.9 respectively. On the 4UNI-Faculty, SYNTHIA achieves a MI score of 66.98 when the compression ratio is 93.53%. The MI-Dual-TLBO achieves the largest MI score which is 68.44. It only outperforms the original web pages by 6.2%. Similar results happen on 4UNI-Student. The MI-Dual-TLBO outperforms all baseline algorithms with a MI score of 67.95. It improves the original web pages by 4.2%. SYNTHIA has a large compression ratio on both 4UNI-Faculty (93.53%) and 4UNI-Student (92.17%). It means
there do not exist many repetitive components which results in that the improvement of MI score is not large.

3.5.3 Effect of Parameters

We evaluate the effect of parameters in our dual-TLBO algorithm using SYNTHIA-DATASET. We evaluate the maximum number of generations, the patience number, the learning rate and the population size. By default, the maximum number of generations is 25, the patience number is 10, the learning rate is 0.9 and the population size equals to 15. As the dual-TLBO generates the initial population randomly, which slight effects the results, we repeat the same experiment 5 times and calculates the average as the final results. We further record the efficiency of every parameters.

Population Size

![Figure 3.13: Effect of population size](image)
The population size is the maximum number of solutions. As the dual-TLBO generates a new solution based on the old solution and replaces the old solution if the new solution has a larger MI score, the population size affects the efficiency of the dual-TLBO. During the mutation phase, we compare the best solution and the worst solution pairwise. We need to evaluate every element of a solution to ensure that every element makes a positive contribution to high MI score. Therefore, if the initial solutions can cover all possible values (0 and 1) of every element, the dual-TLBO is expected to achieve a high MI score.

We test four population sizes: 5, 10, 15 and 20, and the result is shown in Figure 3.13. The blue line represents the MI score, and the orange line shows its corresponding runtime. We scale the time to 1/500 of the original number to show it with MI scores. The x-axis is the population size. The dual-TLBO with a population size of 5 achieves an MI score of 8.43. The MI score grows up to 8.64 when the population size becomes 10. The MI score achieves the peak which is 9.1 when the population size is 20. However, the time efficiency of dual-TLBO also drops dramatically when the population size increases. It increases close 1000 seconds for increasing population size of 5. The experiment result proves that the population size is an important parameter which affects the performance of the dual-TLBO. We argue that the population size should be large when the time efficiency is not important.

**Learning Ratio**

We evaluate the effect of learning rate. We select 0.1, 0.5, 1 and 1.2 as candidates of learning rate. 0.1 is a small number which means the initial population is slightly improved. 1.2 is a large number which means the initial population is significantly improved. The learning rate controls the improvements in every iteration. The dual-TLBO can approach the best solution faster than most of the previous work because every solution improves based on the difference between the teacher and mean multiplied learning rate. Therefore, the learning rate is an important parameter which decides how much improvement that every solution makes during every generation.
The result is shown in Figure 3.14. The blue line is the MI score, and the orange line is the runtime. The x-axis represents the learning rate. The MI score is 8.54 when the learning rate is 0.1, and it grows to 8.79 when the learning rate becomes 0.5. The MI score continues increasing with the increase of learning rate until 8.85. After that, the MI score drops to 8.54 when the learning rate becomes 8.54. The experimental result shows that the learning rate is an important parameter which affects the MI score. The learning rate should between 0.5 and 1 to obtain high MI score. We notice that the runtime is 1546 seconds when the learning rate is 0.1, which is less than the others (about 2000 seconds). We further study this phenomenon and find that the initial solutions generate a smaller number of new solutions compared with other learning ratios. As we round the elements of solutions to 0 and 1, the new solution may be the same as old solutions after learning from teacher and students. It saves time if there does not generate a new solution for comparison and replacement. Also, the best solution may not change after generations, which make the dual-TLBO terminates early (the dual-TLBO stops when it does not generate new best solution after the patience number of generations). It makes
the dual-TLBO have better time efficiency but worse performance in maximizing the MI score than other learning ratios.

**Maximum Number of Generations and Patience Number**

![Figure 3.15: Effect of maximum generations](image_url)

The maximum number of generations is the maximum number of teacher phases and student phases. There exist two termination criterions: the maximum number of generation and patience number. Anyone of the termination criterions is reached then the generations stop. The experimental result of the maximum number of generations is shown in Figure 3.15. The x-axis represents the maximum number of generations, which are 15, 20, 25 and 30 respectively. The blue line is the MI score, and the orange line is the runtime. The result shows that the MI scores are almost the same (around 8.8) and also the runtime (around 2000 seconds). Theoretically, the runtime should increase as the number of generations increase. However, the runtime does not increase much. This is because that dual-TLBO algorithm approaches the best solution quickly and terminates early.
3.5 Experiments

The Figure 3.16 shows the experimental result of the effect of the patience number. We set the patience number like 5, 10, 15 and 20, which is less than the maximum number of generation (25). The MI scores are almost the same (around 8.8) when the patience number increases. Besides, the runtime increases with the patience number. As the MI score does not change dramatically with the patience number, it proves that dual-TLBO can approach the best solution quickly. The dual-TLBO does not need many generations to approach to the best solution.

3.5.4 Time Efficiency

We next evaluate the time efficiency of MI-Dual-TLBO on different size of web pages. The size of a web page is the number of nodes of the web page’s corresponding DOM tree. The experimental results shown in Table 3.5, Table 3.6, Table 3.7, Table 3.8 and Table 3.9 indicate that MI-Dual-TLBO takes longer time when the compression ratio is little. For example, the SYNTHIA takes 2031 seconds for 200 web pages when the compression
ratio is about 75% (SYNTHIA-DATASET). However, the runtime is only 141 seconds for 1124 web pages when the compression ratio is about 93% (4UNI-Student). Intuitively, if there are less repetitive components in one web page, it takes less time to split and fold them. Besides, the web page size also affects the efficiency of our algorithms because the alignment algorithm traverses two sub-trees to capture differences. It should cost a little time when the number of nodes is small. Therefore, we make a statistic of these four data sets respectively and study the relationships between the size of web pages, compression ratio, and MI score. By studying the relationships given above, we can select proper algorithms to apply on various web page data sets, especially when both the efficiency and the maintainability are crucial.

We begin by studying the time efficiency of MI-Dual-TLBO on SYNTIA-DATASET. The statistics result is shown in Figure 3.17. The average runtime of web pages whose size is between 0 and 1000 nodes is 1 second (cf. Figure 3.17(b)), and the average runtime of web pages with 1001 to 3000 nodes is 12 seconds. The average runtime of 3001 to 5000 nodes web pages is about 8 seconds. However, the distribution of the runtime of web pages with 1001 to 5000 nodes is loose, as shown in Figure 3.17(a). The runtime varies in a wide range between 1 second and 80 seconds. When the size of web pages increases to 5001 - 6000, the average runtime is about 23 seconds. The average runtime increases into 43 seconds when the size is 7001 - 8000 nodes. Then, the average runtime decreases when the size is 8001 to 10000 nodes. However, as the number of samples is not as large as the size is 0 - 5000 nodes, the average runtime of large web pages (8001 - 10000 nodes) cannot be general results. However, we can see from the Figure 3.17(a) that the minimum MI runtime increases from 1 second (the web page size is 1000 nodes) to 28 seconds (the web page size is 10000 nodes) when the size of web pages increases. Therefore, the runtime increases when the size of the web page increases in general. When the compression ratio is about 65%, most of the runtime of every web page is below 50 seconds.

The statistics of TBDW is shown in Figure 3.18. It can be seen from the Figure 3.18(a) that most of the web pages can be processed within 20 seconds. For the web pages with size
3.5 Experiments

Figure 3.17: Statistics of SYNTHIA-DATASET

(a) Runtime Distribution of SYNTHIA-DATASET

(b) Average Runtime of SYNTHIA-DATASET
Figure 3.18: Statistics of TBDW

(a) Runtime Distribution of TBDW

(b) Average Runtime of TBDW
between 0 and 1000, MI-Dual-TLBO achieves an average runtime of 3 seconds (cf Figure 3.18(b)). However, it increases to 20 seconds when the size increases to 1000 - 2000. It is significantly larger than other ranges (5 seconds for 2001 - 3000 and 6 seconds for 3001 - 4000). The reason is there exist some web pages whose size is between 1001 and 2000 which cost much more time than others (65 seconds to 100 seconds, cf. 3.18(a)). These web pages takes up a small percentage. The MI-Dual-TLBO thus can have good performance (the runtime is no more than 20 seconds) when the compression ratio is around 75%.

We also study the results of UW-CAN which have a compression ratio of approximate 83%. Most of the web pages have a size of 0 - 1000 (cf. Figure 3.19(a)), and the average runtime is about 1 second. The average runtime increase dramatically to 24 seconds where the size of web pages is 1001 - 2000. The reason is the runtime distributes in a large range which is between 1 second and 90 seconds. The result on web pages whose size are bigger than 2001 is not meaningful (cf. Figure 3.19(b)) as the number of samples is too little (cf. Figure 3.19(a)). Based on the Figure 3.19(a), we can have a conclusion that most of web pages can be processed within 20 seconds when the compression ratio is about 83%.

Finally, we analyze the results on 4Universities. We combine the results from 4UNI-Faculty and 4UNI-Student and analyze them together. The compression ratio of 4Universities is 93%. The size of most of web pages of 4Universities is between 0 - 2000 (cf. Figure 3.20(a)). For web pages which have 0 - 1000 nodes, the average runtime is 1 second. For web pages which have 1001 - 2000 nodes, the average runtime is 2 second.

Considering the results on all datasets, we can see that MI-Dual-TLBO is slower when the compression ratio is less. When there exist many repetitive components, MI-Dual-TLBO takes a long time to process them.
Figure 3.19: Statistics of UW-CAN

(a) Runtime Distribution of UW-CAN

(b) Average Runtime of UW-CAN
3.5 Experiments

(a) Runtime Distribution of 4Universities

(b) Average Runtime of 4Universities

Figure 3.20: Statistics of 4Universities
3.6 Summary

We proposed and studied a web page separation problem that aims to extract easily maintainable template code and data records from web pages. We showed the NP-hardness of the problem and presented a heuristic algorithm to solve the problem. We defined a sub-problem named “Finding Chunk Boundaries” and proposed a dual-teaching and learning based optimization algorithm to solve the sub-problem. The experimental results show that our algorithms outperform state-of-the-art web page separation techniques by 24.4% in the maintainability measure. Our algorithms extracts easily maintainable template code from web pages, Web developers can compare their web page source code with the template code generated by our algorithms and identify ways to improve the maintainability of their source code.
Chapter 4
Web Page Separation for High-quality Data Record

4.1 Introduction

In this chapter, we focus on extracting data records from a set of web pages which are generated from different unknown templates. The structured data records that extracted from web pages can provide data for many important applications, such as stock market prediction [107] and personalized recommendation systems [63].

Most of previous data record extraction work [14,49] assume that the input web pages are generated by the same template. However, recognizing web pages generated by the same template is time-consuming. We lift this assumption and work on extracting data records from a set of web pages generated by different unknown templates. Besides, we focus on extracting the data records from web pages and structure them instead of deducing the schema that provides the data records, which is different from page-level data extraction [49].

Our solution framework processes the web pages with four steps: web page template and data separation, template clustering, template alignment, and data record filtering. The web page separation step separates every web page into template code and data records. The template clustering step clusters the set of web pages by the similarity of template code. The template clustering step aims to group web pages which are generated by the similar template code. The template alignment step then aligns the template
code of web pages in the same group to generate a generalized template code which can generate all web pages in the same group. The generalized template code is utilized to help to extract the data records which is incorrectly extracted during the web page separation step. We detail the framework in Section 4.4.

We further propose a semantic similarity measurement which utilizes sophisticated semantic similarity measurement methods, such as doc2vec [58] and topic modeling [102], and the structure of templates to measure the similarity among templates and data records populated into. The semantic similarity of templates is used in the alignment process of the web page template and data separation steps.

The contributions of this chapter are summarized as follows:

- We formulate a web page separation problem that aims to extract data records from a set of web pages generated by different unknown template.

- We propose a framework, which consists of four steps: web page template and data separation, template clustering, template alignment and data record filtering, to tackle the problem.

- We propose a semantics-aware web page template and data separation algorithm which utilizes the semantic similarity of data records and the structure of templates to measure the similarity between templates and data records populated into.

- We propose a template clustering algorithm that can recognize web pages generated by the same template.

- We conduct an experimental study to evaluate the performance of our framework in data record extraction and recognizing attributes within the data records. Experimental results show that our framework outperforms baseline algorithms which assume a pre-known clustering of the set of web pages in F-Score.

The rest of this chapter is organized as follows: Section 4.2 presents our proposed model and baseline algorithms. Section 4.3 formulates our web page separation problem. Sec-
4.2 Preliminaries

We first describe the model that we used to formulate our problem, in Section 4.2.1. We then give the baseline algorithms MDR and DEPTA in Section 4.2.2, and the baseline algorithm PL+IP+IA in Section 4.2.3.

4.2.1 Model

We parse a web page into a DOM tree. A DOM tree consists of two types of nodes: element nodes and text nodes. An element node consists of a tag and a list of attributes. An attribute is a mapping from an attribute name to an attribute value. A text node is a node with textual content. We represent the template of a web page with a layout tree. Compared with a DOM tree, a layout tree has two additional nodes: reference node and instruction node. When aligning sub-DOM trees or layout trees, if the aligned text nodes or attributes are different, we add a reference node to replace them and record the values in the extracted data records (environments). Instruction node has two types: condition node and loop node. If we consider some sub-DOM trees are similar to each other, we add a loop node as the parent of the sub-layout tree that generated the sub-DOM trees. Besides, if a branch of a sub-DOM tree is not shared by all aligned sub-DOM trees, we add a condition node as the parent of the branch.

The web components of web pages can be classified into general components and data record components. The general components are the web components that do not contain data records, such as header, footer, etc. The data record components wrap data records provided by users, which is usually stored in a database. Although the general components may contain repetitive web components, such as the navigation bar in a web page, the content is a part of the template because they are the same in all web pages gener-
Web Page Separation for High-quality Data Record

Figure 4.1: An example of template components

The components surrounded by blue rectangles are general components. They are fixed in every web page and maintained by the same template. The general components are fixed in the template code. The data record components are different in web pages because they are populated with different data records. We locate the data record components and extract data records within them using the extracted template. For example, Figure 4.1 is a web page from www.amazon.com. The components surrounded by blue rectangles are general components. They are fixed in every web page of products. The components surrounded by red rectangles are data record components. They contain the data records (details about products) which are different in every web page, and the data records are provided by www.amazon.com.

4.2.2 MDR and DEPTA

MDR and DEPTA are techniques which aim to tackle the page-level data extraction problem [49]. They aim to extract data records from a set of web pages which are generated by the same unknown template. MDR is a framework which consists of two steps: mining data regions and data record identification. MDR utilizes tree similarity algorithms to locate data regions in a web page. The data region is an area in the web page populated with data records. MDR considers the data regions in a web page as a table, and every
data region is a row of the table. After recognizing data regions, MDR identifies the data records within every data region. The DEPTA is based on MDR and aligns the extracted data records. The DEPTA is a framework which consists of three steps: mining data regions, mining data records, and data item alignment. DEPTA also considers the data regions as a table. Compared with MDR, DEPTA utilizes visual cues to mine data regions. DEPTA then utilizes tree edit distance \cite{112} and simplified tree matching algorithm \cite{90} to mine data records. DEPTA finally utilizes the partial tree alignment algorithm to align data records.

Compared with MDR and DEPTA, our framework utilizes a global alignment algorithm to align template and data records. Besides, we extract the data records based on the templates of web pages. Besides, our problem is different from the page-level data extraction because we lift the assumption that the set of web pages are generated by the same unknown template.

4.2.3 PL+IP+IA

PL+IP+IA tackles the page-level data extraction problem. PL+IP+IA takes a set of web pages as the input and data records in one of the web pages is labeled with entity names. It aims to extract data records in the other web pages that have the same entity names. PL+IP+IA utilizes multiple features, such as tree structure and semantics of data records, to align data records in other web pages with the data records of the labeled web page. Compared with data record recognition, it requires to align the data records by their semantics. The data records that have similar semantics are aligned. For example, the prices of products and the names of products are aligned respectively. To recognize the data records in web pages, PL+IP+IA utilizes layout features, content features and context features to align data records. The layout features include visual size, visual location and DOM path (tree structure). The content features include the semantics of data records and the format of strings. The context features are the prefix or suffix of string.
4.3 Problem Definition

We aim to extract data records from web pages which are generated by different unknown templates. Our work is a balance of two important aspects of web information extraction, which are web page separation problem (given in SYNTHIA [73]) and data record extraction problem. Compared with page-level data extraction problem (given in FivaTech [49]), we focus on extract data records instead of deducing the schema that provides the data records. We formulate our web page separation problem:

**Definition 4.1.** Given a set of DOM trees $T_1, T_2, \ldots, T_n$, which are generated by different unknown templates, the number of unknown templates is no more than the number of DOM trees, find a method that separates the list of DOM trees into templates and data records that the web pages can be reconstructed by running the templates on the data records.

In this problem, we not only recognize data records in web pages but also structure the data records. We need to align the data values which have similar semantics. For example, in Figure 4.1, the data values “EVGA GeForce ...”, “Gigabyte ...” and “CYBERPOWERPC ...” are the names of products, and they should be aligned.

4.4 Our Method

In this section, we first propose our framework in Section 4.4.1. We then show the semantic similarity measurement that measures the similarity among templates in Section 4.4.2. We next show the web page template and data separation algorithm in Section 4.4.3 and the template clustering and alignment algorithm Section 4.4.4 respectively. We finally give the data record filtering algorithm in Section 4.4.4.

4.4.1 General Framework

The workflow of our framework is shown in Figure 4.2. It consists of four main steps: web page template and data separation, template clustering, template alignment, and record filtering. Web page template and data separation step separates every web page into templates and data records. Then, the template clustering step groups web pages
by templates. This step aims to recognize web pages generated by the same template. Our framework then aligns the templates of web pages in every group to capture the difference among templates to construct a generalized template which can generate all web pages in the same group. The record filtering step processes web pages to extract the data records which is not correctly extracted by the web page template and data separation step. The web page template and data separation step may not extract all data records and align them correctly. One case is that one web page may contain only one data record. In this case, it is hard to extract the data record because it may be a part of the template. Intuitively, template wraps repetitive data records to generate a web page. The web page instead of repetitive web components can be considered as generated by hard coding.

![Workflow of the general framework](image)

**Figure 4.2**: Workflow of the general framework
4.4.2 Semantic Similarity Measurement

To align data records with similar semantics, we need to measure the semantic similarity between data values. We utilize four different ways to measure the semantic similarity between data values, which are latent Dirichlet allocation [10], word2vec [37], word mover distance [56] and doc2vec [58]. These methods are sophisticated semantic similarity measurement applied to non-stop English words. The data values may not be non-stop English words, such as phone numbers and email addresses. These data values usually contain numbers and special cases. We further utilize the formats of data values to measure the semantic similarity. We first describe how to measure the semantic similarity between data values which are English words, and then the measurement applied on data values which are non-English words.

Latent Dirichlet allocation

Latent Dirichlet allocation is a statistics-based algorithm which allocates documents to a number of topics. Let the number of topics be \( K \), the number of vocabulary be \( V \), the number of documents be \( M \), the total number of words in all documents be \( N \), the number of words in document \( d \) is \( N_d \), the prior weight of topic \( k \) in a document be \( \alpha_k \), the prior weight of word \( w \) in a topic be \( \beta_w \), the probability of word \( w \) occurring in topic \( k \) be \( \varphi_{kw} \), the probability of topic \( k \) occurring in document \( d \) is \( \theta_d \), the distribution of topics in document \( d \) be \( \Theta_d \). \( Z \) is the identity of the topic of all words in all documents, and \( W \) is the identity of all words in all documents. The total possibility of the model is as follows:

\[
P(W, Z, \Theta, \varphi, \alpha, \beta) = \prod_{i=1}^{K} P(\varphi_i; \beta) \prod_{j=1}^{M} P(\theta_j; \alpha) \prod_{t=1}^{N} P(Z_{jt} | \theta_j) P(W_{jt} | \varphi Z_{jt})
\]

We utilize the equation to get the distribution of all topics of every document.

\[
\int_{\Theta} \prod_{t=1}^{N} P(Z_{jt} | \theta_j) P(W_{jt} | \varphi Z_{jt}) d\theta_j
\]

We consider every data record as a document. The number of topics is the sum of the number of distinct paths from the leaves to the root of every sibling that are aligned. The
path does not include instruction nodes. For example, if a path is “for[1]/li[1]/a[1]”, we filter out the instruction node and the path is “li[1]/a[1]”. The sum of prior weights of topics of a document is 1. For the $k$ topic of the $k$th document, we first assign the weight of other topics 0.01, and we then calculate the weight by one minus other topics. For example, there are five topics, the prior weight of topic distribution of the first document (the path to the data record maps to the first unique path) is [0.96, 0.01, 0.01, 0.01, 0.01]. We utilize the prior weights and the latent Dirichlet allocation to calculate the distribution of topics of every document, which is a tensor. We utilize the cosine similarity of the distribution of topics of the document to calculate the semantic similarity between data values.

**Word2vec**

Word2vec is a word embedding method which converts a word into a vector. It is widely used in semantic similarity measurement. As every word is represented by a vector, word2vec cannot be directly used to measure the similarity between two data values if the data values are not the same length. Therefore, we calculate the mean vector of words as the vector of the document. The cosine similarity measures the similarity between vectors and returns a number between 0 and 1 to represent the semantic similarity. We further utilize two techniques which are based on word2vec, word mover distance, and doc2vec, to measure the similarity between two data values.

**Word mover distance**

Word mover distance utilizes word embedding to calculate the semantic similarity between two documents. The word mover distance between the two documents is the minimum distance that travels all words in one document into all words in the other document. The word mover distance only considers non-stop words in documents. The distance between two words is a number between 0 and 1. Given two documents, let the number of non-stop words of the first document be $m$, and the number of non-stop words of the second document be $n$, the word mover distance between the two documents is
If $m$ is not equal to $n$, the mismatch of non-stop words results in moving one word to multiple words in the other document. Therefore, the average word mover distance as $\frac{w}{\max(m,n)}$. The semantic similarity between the two documents is $1 - \frac{w}{\max(m,n)}$.

**Doc2vec**

The doc2vec is based on word2vec. The goal of doc2vec is generating tensors which have the same length to represent documents of different length. We utilize the doc2vec to convert data values into vectors. We calculate the cosine similarity between two vectors as the semantic similarity between two data values.

**Format of data record**

If the word is not a non-stop English word, which contains the number and special cases, we utilize the format of the word to calculate the similarity between words. This is designed for measuring the semantic similarity of documents which cannot be measured by the previous methods, such as emails, phone numbers, etc. We utilize a longest common subsequence algorithm to extract the common parts of strings. If two strings have common parts and the same format, e.g., number + “t”, we consider two strings as having the same semantic similarity. The format consists of three parts: digits, characters, and punctuation (e.g., “-”, “|” and “?”). We convert the data value into the format given as above. For example, a phone number “123-456-789” has the format “number-number-number”. If the format of data values are the same, we consider them have the same semantic similarity, and the semantic similarity score is 1.

We use the semantic similarity measurement algorithm given in this section in the following semantics-aware web page separation algorithm.

### 4.4.3 Semantics-aware Web Page Separation Algorithm

We propose a semantics-aware web page separation algorithm for the web page template and data separation procedure. Our separation algorithm separates a web page
4.4 Our Method

into template code and data records. We parse a web page into a DOM tree and process the DOM tree in a bottom-up manner with three steps: splitting, folding and alignment. The splitting algorithm splits siblings into chunks. The siblings in the same chunk contain data records with similar semantics. The siblings whose roots have the same tag and the same attribute names contain data records with similar semantics. The folding algorithm captures the same parts among siblings and adds an instruction node to show that the siblings are folded. The alignment algorithm captures the differences among siblings and adds instruction nodes to show the branches that are not shared by all siblings. Although we discuss the folding algorithm and alignment algorithm separately, the folding algorithm and alignment algorithm concurs while processing the DOM tree. The folding algorithm and the alignment algorithm captures the differences and commons at the same time.

![A DOM tree](image1)

![A Layout Tree](image2)

Figure 4.3: A running example of overall algorithm

Listing 4.1: An environment

```python
1 "for_1": [
2   {"var1": "Apple", "var2": "$4.5"},
3   {"var1": "Banana", "var2": "$8.5", "if_1": {"var3": "Popular"}}
```
To help understand our general process, we give a running example which is shown in Figure 4.3 and Listing 4.1. Our algorithm separate the DOM tree (cf. Figure 4.3(a)) into template code (presented as a layout tree, cf. Figure 4.3(b)) and an environment (cf. Listing 4.1). There are three siblings whose tag is “ul”. The first two siblings are information on fruits. The third sibling shows the information of the front-end development library (“Powered by ReactJS”). Therefore, we fold and align the first two siblings (covered by the yellow rectangle in Figure 4.3) because they have similar semantics in comparison with the third sibling. The folding algorithm adds a loop node (“\langle for_1\rangle”) to instruct that its children are generated by folding siblings. The alignment algorithm adds an option node (“\langle if_1\rangle”) to instruct that its children are not shared by all folded siblings (“\langle ul\rangle/\langle li\rangle/Powered by ReactJS”). The differences of fruit name (“Apple” and “Banana”), price (“$4.5” and “$8.5”) is captured by adding reference nodes (”\langle var1\rangle” and “\langle var2\rangle”). We also record the text nodes in the environment (the variables “var1” and “var2”, cf. Listing 4.1). Also, we record the text records in the optional branches (cf. the branch “\langle if_1\rangle” in Figure 4.3(b)). We add a reference node (“\langle var3\rangle”, cf. Figure 4.3(b)) to replace the text node (“Powered by ReactJS”, cf. Figure 4.3(a)). The environment also records the variable (“if_1/var3”). This is different from the environment which is defined in SYNTHIA [73]. In SYNTHIA, the environment only records the text nodes which can be found in multiple siblings. The “\langle var3\rangle” is not added according to SYNTHIA. However, as our task is to extract data records from web pages, we add the reference node (“\langle var3\rangle”) when the text node is not shared by multiple siblings.

We then describe the folding algorithm. The folding algorithm aims to capture the common parts among siblings. For the element node, if the attribute names and tag name are the same, the folding algorithm keeps it in the layout tree. For the text node, the folding algorithm keeps it in the layout tree when the values of text nodes are the same. Besides, the folding algorithm adds a loop node, whose tag name consists of “for” and its loop node index, to show that its children are folded by several siblings.
We next present the alignment algorithm. To align two layout trees, we begin with aligning the path from the roots of two layout trees to their leaf nodes. For two element nodes, if its tag name and attribute names are the same, we consider they are equal element nodes. For two text nodes, we consider they are equal text nodes while aligning even if their values are different. For example, a text node “apple” and a text node “grape” is equal, while their values are different. If the types of two nodes are different, e.g., one is an element node, and the other is a text node, they are node equal nodes. If every node in two paths is the same, we consider the two paths as equal to each other. Equal paths can be aligned. Therefore, there exist many potential methods to align two layout trees. We choose the alignment which can structure data records. We achieve it by evaluating the semantic similarity between data records and group data records with similar semantics.

We then show how to measure the semantic similarity between two reference nodes or text nodes. Let two text be $t_i$ and $t_j$. A semantic similarity $TS(t_i, t_j)$ calculates the semantic similarity between $t_i$ and $t_j$. There exist many ways to calculate the semantic similarity between two text items, such as topic modeling [102], word2vec [37] and doc2vec [58]. The semantic similarity function $TS(t_i, t_j)$ returns a number between 0 and 1. We consider the following cases when measuring the semantic similarity between two leaf nodes $n_i$ and $n_j$ ($n_i$ and $n_j$ are text nodes or reference nodes or null):

- If both $n_i$ and $n_j$ are text nodes, the semantic similarity between $n_i$ and $n_j$ is $TS(n_i, n_j)$.
- If $n_i$ is a reference node, and $n_j$ is a text node, the semantic similarity between $n_i$ and $n_j$ is the smallest semantic similarity between all text nodes referred by $n_i$ and $n_j$.
- If both $n_i$ and $n_j$ are reference nodes, the semantic similarity between $n_i$ and $n_j$ is the smallest semantic similarity between all text nodes referred by $n_i$ and all text nodes referred by $n_j$.
- If $n_i$ is a text node and $n_j$ is null, which means there does not exist a corresponding node of $n_i$ in the other sibling, the semantic similarity between $n_i$ and $n_j$ is 0.
• If \( n_i \) is a reference node and \( n_j \) is null, which means there does not exist a corresponding node of \( n_i \) in the other sibling, the semantic similarity between \( n_i \) and \( n_j \) is the smallest semantic similarity between every two pair of text nodes referred by \( n_i \).

To help understand the similarity between two leaf node, we give examples to explain, which is shown as follows:

• If \( n_i \) and \( n_j \) are text nodes, the value of \( n_i \) is “apple” and the value of \( n_j \) is “banana”, let \( TS(n_i, n_j) \) be 0.58, the semantic similarity between \( n_i \) and \( n_j \) is 0.58.

• If \( n_i \) and \( n_j \) are reference nodes, let \( n_i \) refer to “apple” and “banana”, \( n_j \) refers to “grape” and “mango”, and \( TS(“apple”, “grape”) \) is 0.52, \( TS(“apple”, “mango”) \) is 0.61, \( TS(“banana”, “grape”) \) is 0.51, \( TS(“banana”, “mango”) \) is 0.74, \( TS(n_i, n_j) \) is 0.51 which is the smallest number among all semantic similarity between all text nodes referred by \( n_i \) (“apple” and “banana”) and all text nodes referred by \( n_j \) (“grape” and “mango”).

• If \( n_i \) is a text node and its value is “apple”, \( n_j \) is a reference node which refers to “grape” and “mango”, \( TS(“apple”, “mango”) \) is 0.61, \( TS(“apple”, “grape”) \) is 0.52, the semantic similarity between \( n_i \) and \( n_j \) is \( TS(n_i, n_j) = 0.52 \). The smallest semantic similarity between \( n_i \) and all text nodes referred by \( n_j \) is 0.52 (the similarity score between “apple” and “grape”).

• If \( n_i \) is a text node and its value is “apple”, and \( n_j \) is null, the semantic similarity between \( n_i \) and \( n_j \) \( TS(n_i, n_j) \) is 0.

• If \( n_i \) is a reference node which refers to text nodes “apple”, “grape” and “mango”, and \( n_j \) is null, let \( TS(“apple”, “mango”) \) be 0.61, \( TS(“apple”, “grape”) \) be 0.52, \( TS(“grape”, “mango”) \) be 0.56, the semantic similarity score between \( n_i \) and \( n_j \) \( TS(n_i, n_j) \) is 0.52, because \( TS(“apple”, “grape”) \) is the smallest pairwise semantic similarity among text nodes referred by \( n_i \).

The semantic similarity between two layout trees is the semantic similarity between the aligned layout tree and null. The alignment algorithm aims to maximize the semantic
similarity between two layout trees. When aligning two layout trees, the sequence of siblings are maintained because our framework works in a lossless manner.

The pseudo code (Algorithm 10) summarizes the alignment algorithm. The alignment algorithm begins with initializing a matrix $\text{ALIGN}$ with a size of $(S_p \cdot \text{len} + 1) \times (S_q \cdot \text{len} + 1)$ (Line 2). For the first element of every row and column, we set an initial value, which is the negative index (Lines 3 - 6). Then, the alignment algorithm starts a recursive loop (Lines 8 - 9) which try the alignment of subsequence of siblings. For example, the element $\text{ALIGN}[i, j]$ is the maximum semantic similarity of aligning $S_p[1 : i]$ (from the first sibling to the $i$th sibling of $S_p$) and $S_q[1 : j]$ (from the first sibling to the $j$th sibling of $S_q$).

For every subsequence of $S_p$ and $S_q$, the alignment algorithm tries to (1) align $S_{pi}$ and $S_{qj}$ (Line 10), (2) align $S_p[1 : i - 1]$ and $S_q[1 : j]$ and mismatch $S_{pi}$, and (3) align $S_p[1 : i]$ and $S_q[1 : j - 1]$ and mismatch $S_{qj}$. If $S_{pi}$ and $S_{qj}$ can be aligned, $\text{TS}(S_{pi}, S_{qj})$ returns the semantic similarity of $S_{pi}$ and $S_{qj}$; otherwise, it returns $1 - \text{TS}(S_{pi}, \text{null}) + 1 - \text{TS}(\text{null}, S_{qj})$, which means both $S_{pi}$ and $S_{qj}$ are mismatched. $1 - \text{TS}(S_{pi}, \text{null})$ shows that although $S_{pi}$ is mismatched, and the penalty of mismatching is 1, $S_{pi}$ adds to the semantic similarity.

Therefore, the alignment minus $1 - \text{TS}(S_{pi}, \text{null})$. Following this algorithm, we minus $1 - \text{TS}(\text{null}, S_{qj})$ when $S_{qj}$ is mismatched. We select the choice with the maximum semantic similarity, and save it to $\text{ALIGN}[i, j]$ in Lines 13 - 18. The alignment algorithm then backtracks according to $\text{ALIGN}$ and returns a matched list $\text{Aligned\_Sequence}$. If $\text{ALIGN}[i, j] == \text{ALIGN}[i - 1, j - 1] + \text{TS}(S_{pi}, S_{qj})$, $S_{pi}, S_{qj}$ are matched. Therefore, the alignment algorithm adds the pair $(S_{pi}, S_{qj})$ to the matched list $\text{Aligned\_Sequence}$ (Lines 24 - 27). If $\text{ALIGN}[i, j] = \text{ALIGN}[i - 1, j] + (1 - \text{TS}(S_{pi}, \text{null}))$, $S_{pi}$ mismatches the siblings in $S_q[1 : j]$. Therefore, we add $(S_{pi})$ to $\text{Aligned\_Sequence}$ (Lines 28 - 30 and Lines 35 - 36). If $\text{ALIGN}[i, j] = \text{ALIGN}[i, j - 1] + (1 - \text{TS}(\text{null}, S_{qj}))$, $S_{qj}$ mismatches siblings in $S_p[1 : i]$ (Lines 31 - 33 and Lines 38 - 39). Finally, the alignment algorithm returns $\text{Aligned\_Sequence}$ as the result of alignment (Line 40).
Algorithm 10: Alignment

**Data:** the first-level children $S_p$ of $T_p$, the first-level children $S_q$ of $T_q$

1 // Initialization;
2 initialize a matrix $ALIGN[S_p, len + 1, S_q, len + 1]$;
3 for $i \leftarrow 1$ to $S_p, len + 1$ do
4     $ALIGN[i, 0] \leftarrow -i$;
5 for $j \leftarrow 1$ to $S_q, len + 1$ do
6     $ALIGN[0, j] \leftarrow -j$;
7 // Alignment;
8 for $i \leftarrow 1$ to $S_p, len + 1$ do
9     for $j \leftarrow 1$ to $S_q, len + 1$ do
10     ms = $ALIGN[i - 1, j - 1] + TS(S_{pi}, S_{qj})$;
11     ps = $ALIGN[i - 1, j] - (1 - TS(S_{pi}, null))$;
12     qs = $ALIGN[i, j - 1] - (1 - TS(null, S_{qj}))$;
13     if $ms \geq ps$ and $ms \geq qs$ then
14         $ALIGN[i, j] = ms$;
15     else if $ps \geq qs$ then
16         $ALIGN[i, j] = ps$;
17     else
18         $ALIGN[i, j] = qs$;
19 // Backtrack;
20 initialize a list $Aligned\_Sequence$;
21 $i = S_p, len$;
22 $j = S_q, len$;
23 while $i > 0$ or $j > 0$ do
24     if $ALIGN[i, j] == ALIGN[i - 1, j - 1] + TS(S_{pi}, S_{qj})$ then
25         add $(S_{pi}, S_{qj})$ to the beginning of $Aligned\_Sequence$;
26         $i = i - 1$;
27         $j = j - 1$;
28     if $ALIGN[i, j] = ALIGN[i - 1, j] + (1 - TS(S_{pi}, null))$ then
29         add $(S_{pi})$ to the beginning of $Aligned\_Sequence$;
30         $i = i - 1$;
31     if $ALIGN[i, j] = ALIGN[i, j - 1] + (1 - TS(null, S_{qj}))$ then
32         add $(S_{qj})$ to the beginning of $Aligned\_Sequence$;
33         $j = j - 1$;
34 while $i > 0$ do
35         add $(S_{pi})$ to the beginning of $Aligned\_Sequence$;
36         $i = i - 1$;
37 while $j > 0$ do
38         add $(S_{qj})$ to the beginning of $Aligned\_Sequence$;
39         $j = j - 1$;
40 return $Aligned\_Sequence$;
4.4 Our Method

4.4.4 Template Clustering and Alignment Algorithm

The template clustering algorithm aims to group web pages which are generated by the similar template code. The template alignment algorithm aligns the template code of web pages in the same group to generate a generalized template code which can generates all web pages in the same group. Although we describe the template clustering algorithm and the template alignment algorithm separately, they work together in application. We utilize the same alignment algorithm, which is described in the last Section 4.4.3, to align the template codes. The template clustering algorithm is summarized in the pseudo code (Algorithm 11).

Algorithm 11: Template Clustering Algorithm

Data: A list of layout trees \( L = l_1, l_2, ..., l_k \), threshold \( \delta \)

1. sort \( L \) by the number of data records in a descending order;
2. \( clusters = [] \);  
3. while \( L \) is not empty do  
4. \( l = L.pop() \);  
5. \( cluster = [l] \);  
6. \( index = 1 \);  
7. while \( index <= L.len \) do  
8. if \( \text{Similarity}(l, L[\text{index}]) > \delta \) then  
9. add \( L[\text{index}] \) to \( cluster \);  
10. update \( l \) by aligning \( l \) with \( L[\text{index}] \);  
11. remove \( L[\text{index}] \) from \( L \);  
12. add \( cluster \) to \( clusters \);  
13. return \( clusters \);

As shown in Algorithm 11, we store the template code of web pages (stored in layout trees) in a list and set a threshold which decides whether two layout trees should be in the same group. The template clustering algorithm sorts the page by the number of data records in descending order (Line 1). One challenge in page-level data record extraction is extracting data records web pages which have only one data record. We assume that not all web pages have only one data record. The template clustering algorithm begins from the web page which has the most data records. The web page has the most data records is likely to locate the data records correctly. The template clustering algorithm...
then initializes a variable $clusters$ which stores the clustering results (Line 2). The template clustering algorithm next begins a loop (Line 3). While there are layout trees in $L$, the first layout tree in $L$ is pop (Line 4) and inserted into a temporary cluster $cluster$ (Line 5). From the first layout tree in the cluster (Line 6), we calculate the similarity score between $l$ and $L[index]$ and insert the layout tree to $cluster$ if the similarity score between $l$ and $L[index]$ is bigger than the threshold $\delta$ (Line 8). Besides, $l$ and $L[index]$ is aligned and update $l$ to the layout tree after alignment (Line 10). The alignment algorithm is describe in last section (Section 4.4.3). $L[index]$ is removed from $L$ (Line 11). After the inner loop, we add $cluster$ to $clusters$ (Line 12). The template clustering algorithm continues until $L$ is empty. Finally, the template clustering algorithm returns $clusters$ as the clustering results.

We then give how to calculate the similarity between the layout trees. We try to align two layout trees $L_i$ and $L_{i+1}$. Let $L_i$ has more data records than $L_{i+1}$, $L_{i+1}$’s data record components have $N_{i+1}$ nodes (except instruction nodes), and the number of data record components nodes that shared by $L_i$ and $L_{i+1}$ is $N'(i, i + 1)$ (except instruction nodes). The similarity score of $L_i$ and $L_{i+1}$ is $\frac{N'(i, i + 1)}{N_{i+1}}$. For example, if $L_i$’s data record components have ten nodes, and $L_{i+1}$’s data record components have eight nodes, the aligned layout tree of $L_i$ and $L_{i+1}$ have 11 nodes, which means $11 - 10 = 1$ node is uniquely owned by $L_{i+1}$. Therefore, $L_i$ and $L_{i+1}$’s data record components have seven common nodes (the result is from attempting to align $L_i$ and $L_{i+1}$). The similarity score of $L_i$ and $L_{i+1}$ is $\frac{7}{8} = 0.875$. If the similarity score is bigger than a threshold (a number between 0 and 1), $L_i$ and $L_{i+1}$ are grouped.

After clustering web pages by their layout trees, we align every group and output the data records. In application, the template clustering algorithm concurs with the alignment algorithm. If two trees are similar to each other, the aligned layout tree is kept for the next comparison.
4.4.5 Record Filtering

The template clustering and alignment algorithm can extract a generalized template code. We then utilize the generalized template code to verify the extracted data records and reprocess web pages where the data records are not correctly extracted. Given a template code, we consider the web component which wraps the most number of unique data records as the data record component. The reason is the general components may contain repetitive sub-DOM trees, for example, the header with multiple navigation links. However, the general components are usually the same in different web pages, which consists of less unique data records (the navigation links and their titles are the same in different web pages).

The record filtering algorithm locates the data record components in the generalized template code. For every web page, if the data record components in the web page are not covered by the data record components in the generalized template code, we utilize the generalized template code to extract the data records in the web page.

4.5 Experiments

4.5.1 Settings

Data Sets

We utilize the data set named SWDE [42]. SWDE is a read-word collection of web documents which is specifically designed for web information extraction. It contains eight verticals which are auto, book, camera, job, movie, NBA player, restaurant, and university. Every vertical consist of 10 websites and every website has 200-2000 web pages. Every vertical contains 3 - 5 attributes. There is only one attribute value of each attribute on every web page. The verticals is summarized in Table 4.1.

We randomly shuffle the web pages of the same vertical and apply our framework on
Table 4.1: Verticals of SWDE

<table>
<thead>
<tr>
<th>Vertical</th>
<th>Pages</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auto</td>
<td>17923</td>
<td>model, price, engine, fuel</td>
</tr>
<tr>
<td>Book</td>
<td>20000</td>
<td>title, author, isbn, publisher, publication</td>
</tr>
<tr>
<td>Camera</td>
<td>5258</td>
<td>model, price, manufacturer</td>
</tr>
<tr>
<td>Job</td>
<td>20000</td>
<td>title, company, location, date</td>
</tr>
<tr>
<td>Movie</td>
<td>20000</td>
<td>title, director, genre, mpaa</td>
</tr>
<tr>
<td>NBA Player</td>
<td>4405</td>
<td>name, team, height, weight</td>
</tr>
<tr>
<td>Restaurant</td>
<td>20000</td>
<td>name, address, phone, cuisine</td>
</tr>
<tr>
<td>University</td>
<td>16705</td>
<td>name, phone, website, type</td>
</tr>
</tbody>
</table>

every vertical. We make statistics on every attribute and evaluate them respectively. We utilize SWDE as it provides carefully labeled ground-truth for text items in the web pages. Compared with data records extraction, text item extraction is more difficult because one data record usually covers a region of a web page which contains multiple text items.

The second data set we use is TBDW [108]. TBDW contains 51 databases with well-labeled data records of web pages. Every web page contains multiple data records. Compared with SWDE, TBDW label web pages in record-level.

Measurement

We then describe the measurement method named F-Score. F-Score is a common method for testing accuracy. It considers both precision and recall and measures them in one number. For binary classification, F-score considers four data points: true positive, false positive and false negatives. A true positive is a data point that is recognized as positive by our framework, and it is correct. A false positive is a data point that is correct by not recognized by our framework. A false negative is a data point that is recognized as negative by our framework, but it is positive. The precision is calculated by the equation 4.1 and the recall is calculated by the equation 4.1. F-Score is calculated by the equation 4.3.

\[
\text{precision} = \frac{\text{true positives}}{\text{true positives} + \text{false positives}} \tag{4.1}
\]
In our case, the extracted data values which are data records are true positives. The extracted data values which are not data records are false positives. The data records which are not extracted are false positives. For the SWDE dataset, we need to recognize the attributes that we need. As our framework are not designed for entity recognition, we consider the set of data values as the attributes if it meets the following limitations:

- The set of data values have more than two data values.
- At least fifty percent of the data values are one attributes.

For example, a set of data values is “apple, banana, book”. As “apple” and “banana” are fruit, we consider the attribute name as “fruit”. As the “book” is not a fruit, the precision of extracting the attribute “fruit” is $\frac{2}{3}$.

### 4.5.2 Results and Discussion

We first apply our framework on TBDW and compare it with baseline algorithms. The first data record and the links wrapped by other data records construct the ground truth. We compare our framework with MDR [4], DEPTA [109] and SYNTHIA [73]. The result of MDR, DEPTA, and SYNTHIA is reported in the SYNTHIA paper [73]. The result is shown in Table 4.2. We name our framework by Semantics-Aware Web page separation (SAW) As shown in Table 4.2, we denote our framework by “SAW”. MDR achieves a precision of 0.93 and a recall of 0.61. The F-Score of MDR is 0.74. DEPTA achieves a precision of 0.98 and a recall of 0.54. The F-Score of DEPTA is 0.70. The SYNTHIA achieves a precision of 0.99 and a recall of 0.96. The F-Score of SYNTHIA is 0.97. Our framework which utilizes LDA as the semantic similarity function (SAW (LDA)) achieves a precision of 0.99 and a recall of 0.94. The F-Score of SAW (LDA) is 0.96. Our framework which
Table 4.2: Experimental Results on TBDW

<table>
<thead>
<tr>
<th>Method</th>
<th>Precision</th>
<th>Recall</th>
<th>F-Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>MDR</td>
<td>0.93</td>
<td>0.61</td>
<td>0.74</td>
</tr>
<tr>
<td>DEPTA</td>
<td>0.98</td>
<td>0.54</td>
<td>0.7</td>
</tr>
<tr>
<td>SYNTHIA</td>
<td>0.99</td>
<td>0.96</td>
<td>0.97</td>
</tr>
<tr>
<td>SAW (LDA)</td>
<td>0.99</td>
<td>0.94</td>
<td>0.96</td>
</tr>
<tr>
<td>SAW (WM)</td>
<td>0.98</td>
<td>0.94</td>
<td>0.96</td>
</tr>
<tr>
<td>SAW (WV)</td>
<td>0.97</td>
<td>0.95</td>
<td>0.96</td>
</tr>
<tr>
<td>SAW (DV)</td>
<td>0.99</td>
<td>0.97</td>
<td>0.98</td>
</tr>
</tbody>
</table>

Our framework which utilizes word mover distance (SAW (WM)) achieves a precision of 0.98 and a recall of 0.94. The F-Score of SAW (WM) is 0.96. Our framework which utilizes word2vec (SAW (WV)) achieves a precision of 0.97 and a recall of 0.95. The F-Score of SAW (WV) is 0.96. Our framework which utilizes doc2vec (SAW (DV)) achieves a precision of 0.99 and a recall of 0.97. The F-Score of SAW (DV) is 0.98. Our framework which utilizes doc2vec outperforms MDR by 0.24 and DEPTA by 0.28. It also outperforms SYNTHIA by 0.01 in the F-Score measure. However, as reported in the SYNTHIA paper, they first put the web pages generated by the same template in a common root of a DOM tree. It means that they have a pre-known clustering of the set of web pages. In our framework, the clustering is based on the results of the web page template and data separation. Our framework achieves comparable results, which proves that our algorithm can save labor when the web pages are mixed.

Our framework achieves a high precision (0.99) on TBDW dataset. However, the recall is relatively low (0.96). We review the web pages in TBDW and find many nested data records. For example, many web pages utilize “<table>” tag to arrange data records, and also the layout of web pages. As the data records have multiple levels, and the data records in the same level are different, many data records cannot be grouped (relevant data records cannot be grouped). This results in a low recall. Besides, we find that some data records have a different structure with other records in the same web pages. Our framework considers them generated by different templates. This may result in that data records are split into different groups. This case reduces the recall of our framework.
We next evaluate our framework on the SWDE data set. Compared with TBDW, the ground truth of SWDE is labeled in the data item level (every string has a label). This requires that the data record extraction technique can locate not only the data records but also group data items correctly to achieve high performance. We compare our framework with the baseline algorithms given in the original paper of SWDE [42], which is named PL+IP+IA. We utilize the auto dataset which consists of 17923 web pages to evaluate our framework. We denote the data set by SWDE-AUTO. We select to test on SWDE-AUTO because the price, fuel, and engine usually consist of non-English words.

<table>
<thead>
<tr>
<th>Method</th>
<th>Attribute Name</th>
<th>Precision</th>
<th>Recall</th>
<th>F-Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>PL+IP+IA</td>
<td>model</td>
<td>0.46</td>
<td>0.41</td>
<td>0.43</td>
</tr>
<tr>
<td>SAW</td>
<td>model</td>
<td>0.60</td>
<td>0.63</td>
<td>0.61</td>
</tr>
<tr>
<td>PL+IP+IA</td>
<td>price</td>
<td>0.80</td>
<td>0.79</td>
<td>0.80</td>
</tr>
<tr>
<td>SAW</td>
<td>price</td>
<td>0.86</td>
<td>0.82</td>
<td>0.84</td>
</tr>
<tr>
<td>PL+IP+IA</td>
<td>engine</td>
<td>0.82</td>
<td>0.82</td>
<td>0.82</td>
</tr>
<tr>
<td>SAW</td>
<td>engine</td>
<td>0.89</td>
<td>0.85</td>
<td>0.87</td>
</tr>
<tr>
<td>PL+IP+IA</td>
<td>fuel</td>
<td>0.81</td>
<td>0.73</td>
<td>0.77</td>
</tr>
<tr>
<td>SAW</td>
<td>fuel</td>
<td>0.74</td>
<td>0.70</td>
<td>0.72</td>
</tr>
</tbody>
</table>

The experimental result of SWDE-AUTO is shown in table 4.3. Our framework outperforms PL+IP+IA in model, price and engine. However, our framework does not outperforms PL+IP+IA in fuel, but does not achieves the same performance of PL+IP+IA. We find that the structure of fuel is various, such as “19 MPG city / 27 MPG highway” and “28 City / 34 Hwy”. “Hwy” is the abbreviation of “highway”. However, the abbreviation of English words effects the semantics similarity measurement, because there are not many abbreviations of English words. Although price and engine also consists of non-English words, our algorithm also outperforms PL+IP+IA. We review the ground-truth of price and engine, and find that their formats are relatively fixed. For example, the price is like $8,000. Every price begins with a dollar symbol following by a number. Also, there is a comma within the number, because the price is usually bigger than 1000. The engine is like “1.6L I4”, which contains the displacement and power of the auto. Our framework
also performs well on model. The model usually consists of English words, e.g., “Ford SUV”. The model consists of the brand and type of an auto. The semantic similarity functions, such as doc2vec and word mover distance can measure the similarity among these words well. We test our framework on the vertical of book. We denote the dataset as SWDE-BOOK. We select this dataset because the similarity of the title of a book is hard to measure. The title of a book can be a word, a term or a short sentence. Although they can be measured by the semantics similarity function, the result is hard to represent that they are all book title. For example, “Jane Eyre” is a title of a book and also a personal name. Therefore, the semantic similarity between “Pride and Prejudice” and “Jane Eyre” is smaller than the semantic similarity between “Alexandre Dumas”, “Jane Eyre”. The result of SWDE-BOOK is shown in table 4.4. As show in table 4.4 our framework out-

<table>
<thead>
<tr>
<th>Method</th>
<th>Attribute Name</th>
<th>Precision</th>
<th>Recall</th>
<th>F-Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>PL+IP+IA</td>
<td>title</td>
<td>0.89</td>
<td>0.87</td>
<td>0.88</td>
</tr>
<tr>
<td>SAW</td>
<td>title</td>
<td>0.92</td>
<td>0.93</td>
<td>0.92</td>
</tr>
<tr>
<td>PL+IP+IA</td>
<td>author</td>
<td>0.95</td>
<td>0.89</td>
<td>0.92</td>
</tr>
<tr>
<td>SAW</td>
<td>author</td>
<td>0.95</td>
<td>0.90</td>
<td>0.92</td>
</tr>
<tr>
<td>PL+IP+IA</td>
<td>isbn</td>
<td>0.84</td>
<td>0.84</td>
<td>0.84</td>
</tr>
<tr>
<td>SAW</td>
<td>isbn</td>
<td>0.89</td>
<td>0.85</td>
<td>0.87</td>
</tr>
<tr>
<td>PL+IP+IA</td>
<td>publisher</td>
<td>0.81</td>
<td>0.81</td>
<td>0.81</td>
</tr>
<tr>
<td>SAW</td>
<td>publisher</td>
<td>0.85</td>
<td>0.84</td>
<td>0.84</td>
</tr>
<tr>
<td>PL+IP+IA</td>
<td>publication</td>
<td>0.88</td>
<td>0.88</td>
<td>0.88</td>
</tr>
<tr>
<td>SAW</td>
<td>publication</td>
<td>0.91</td>
<td>0.90</td>
<td>0.90</td>
</tr>
</tbody>
</table>

performs PL+IP+IA in all attributes. We find that many books have a person’s name. However, titles and authors are wrapped by different web components, which help distinguish the similar attributes. As our framework evaluates the similarity between data records by both semantic similarity and structural similarity, our framework is effective to distinguish data records which have similar semantics.

We test our framework on the vertical of cameras. We denote the data set by SWDE-CAMERA. The result is shown in table 4.5. Our framework achieves similar performance with PL+IP+IA. The model and manufacturer usually contains the brand of the camera,
Table 4.5: Experimental Results on SWDE-CAMERA

<table>
<thead>
<tr>
<th>Method</th>
<th>Attribute Name</th>
<th>Precision</th>
<th>Recall</th>
<th>F-Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>PL+IP+IA</td>
<td>model</td>
<td>0.93</td>
<td>0.88</td>
<td>0.90</td>
</tr>
<tr>
<td>SAW</td>
<td>model</td>
<td>0.93</td>
<td>0.90</td>
<td>0.91</td>
</tr>
<tr>
<td>PL+IP+IA</td>
<td>price</td>
<td>0.98</td>
<td>0.90</td>
<td>0.94</td>
</tr>
<tr>
<td>SAW</td>
<td>price</td>
<td>0.96</td>
<td>0.93</td>
<td>0.94</td>
</tr>
<tr>
<td>PL+IP+IA</td>
<td>manufacturer</td>
<td>0.96</td>
<td>0.93</td>
<td>0.94</td>
</tr>
<tr>
<td>SAW</td>
<td>manufacturer</td>
<td>0.95</td>
<td>0.94</td>
<td>0.94</td>
</tr>
</tbody>
</table>

such as Canon and Samsung. Therefore, our semantic similarity function can effectively recognize words with similar semantics. Also, compared with the manufacturer, the model usually consists of extra properties of a camera, e.g., f2.0. Therefore, it is easy to distinguish the manufacturer from the model.

We test our framework on the vertical of jobs. We denote the data set by SWDE-JOB.

Table 4.6: Experimental Results on SWDE-JOB

<table>
<thead>
<tr>
<th>Method</th>
<th>Attribute Name</th>
<th>Precision</th>
<th>Recall</th>
<th>F-Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>PL+IP+IA</td>
<td>title</td>
<td>0.99</td>
<td>0.93</td>
<td>0.95</td>
</tr>
<tr>
<td>SAW</td>
<td>title</td>
<td>0.96</td>
<td>0.91</td>
<td>0.93</td>
</tr>
<tr>
<td>PL+IP+IA</td>
<td>company</td>
<td>0.84</td>
<td>0.80</td>
<td>0.82</td>
</tr>
<tr>
<td>SAW</td>
<td>company</td>
<td>0.80</td>
<td>0.80</td>
<td>0.80</td>
</tr>
<tr>
<td>PL+IP+IA</td>
<td>location</td>
<td>0.87</td>
<td>0.84</td>
<td>0.85</td>
</tr>
<tr>
<td>SAW</td>
<td>location</td>
<td>0.89</td>
<td>0.85</td>
<td>0.87</td>
</tr>
<tr>
<td>PL+IP+IA</td>
<td>company</td>
<td>0.79</td>
<td>0.77</td>
<td>0.78</td>
</tr>
<tr>
<td>SAW</td>
<td>company</td>
<td>0.75</td>
<td>0.78</td>
<td>0.76</td>
</tr>
</tbody>
</table>

The result is shown in table 4.6. Our framework does not outperforms PL+IP+IA in all attributes except for the location. The reason is the location usually consists of a city or a country, for example, USA - New York. Our semantic similarity functions can recognize the locations well. However, the format and cases of other attributes are various. For example, although the title of jobs are appeared in every data record of job and usually the first data item, our framework cannot group them well. The reason is there are many special cases in the job titles, such as C# programmer and SQL & Security Manager. These special cases and various formats effect the recognition of data items with similar semantics.
We test our framework on the vertical of movies. We denote the data set by SWDE-MOVIE. The experimental result is shown in Table 4.7. Our framework outperforms PL+IP+IA on all attributes, especially the rating. The rating usually contains a character, such as “R” and “N”. Our framework successfully groups many of them together regard less that the rating is in near the end of siblings, which is hard to aligned by pure tree structure similarity. For the names of directors, there exists some non-English names, which effects the semantic similarity measurement.

We test our framework on the vertical of NBA players. We denote the data set by SWDE-NBA. The result is shown in Table 4.8. Our framework outperforms PL+IP+IA in all attributes except for the height. Our framework perform badly in height because there are many formats to show the height, such as “2.16m”, “7ft”, etc. Our semantic similarity
function is not able to tackle these formats. Therefore, the semantic similarity functions return a small number to describe the semantic similarity among weight.

We test our framework on the vertical of restaurants. We denote the data set by SWDE-

Table 4.9: Experimental Results on SWDE-RESTAURANT

<table>
<thead>
<tr>
<th>Method</th>
<th>Attribute Name</th>
<th>Precision</th>
<th>Recall</th>
<th>F-Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>PL+IP+IA</td>
<td>name</td>
<td>0.95</td>
<td>0.89</td>
<td>0.92</td>
</tr>
<tr>
<td>SAW</td>
<td>name</td>
<td>0.93</td>
<td>0.86</td>
<td>0.89</td>
</tr>
<tr>
<td>PL+IP+IA</td>
<td>address</td>
<td>0.97</td>
<td>0.96</td>
<td>0.96</td>
</tr>
<tr>
<td>SAW</td>
<td>address</td>
<td>0.99</td>
<td>0.98</td>
<td>0.98</td>
</tr>
<tr>
<td>PL+IP+IA</td>
<td>phone</td>
<td>1.0</td>
<td>0.98</td>
<td>0.99</td>
</tr>
<tr>
<td>SAW</td>
<td>phone</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
</tr>
<tr>
<td>PL+IP+IA</td>
<td>cuisine</td>
<td>0.98</td>
<td>0.94</td>
<td>0.96</td>
</tr>
<tr>
<td>SAW</td>
<td>cuisine</td>
<td>0.97</td>
<td>0.96</td>
<td>0.96</td>
</tr>
</tbody>
</table>

RESTAURANT. The result is shown in table 4.9. Our framework achieves comparable results with PL+IP+IA. Our framework cannot win PL+IP+IA in the name because of the special cases on the name and non-English names.

We test our framework on the vertical of universities. We denote the data set by SWDE-

Table 4.10: Experimental Results on SWDE-UNIVERSITY

<table>
<thead>
<tr>
<th>Method</th>
<th>Attribute Name</th>
<th>Precision</th>
<th>Recall</th>
<th>F-Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>PL+IP+IA</td>
<td>name</td>
<td>0.97</td>
<td>0.95</td>
<td>0.96</td>
</tr>
<tr>
<td>SAW</td>
<td>name</td>
<td>0.98</td>
<td>0.98</td>
<td>0.98</td>
</tr>
<tr>
<td>PL+IP+IA</td>
<td>phone</td>
<td>0.79</td>
<td>0.78</td>
<td>0.79</td>
</tr>
<tr>
<td>SAW</td>
<td>phone</td>
<td>0.85</td>
<td>0.80</td>
<td>0.82</td>
</tr>
<tr>
<td>PL+IP+IA</td>
<td>website</td>
<td>0.96</td>
<td>0.83</td>
<td>0.89</td>
</tr>
<tr>
<td>SAW</td>
<td>website</td>
<td>0.96</td>
<td>0.91</td>
<td>0.93</td>
</tr>
<tr>
<td>PL+IP+IA</td>
<td>type</td>
<td>0.70</td>
<td>0.68</td>
<td>0.69</td>
</tr>
<tr>
<td>SAW</td>
<td>type</td>
<td>0.71</td>
<td>0.65</td>
<td>0.68</td>
</tr>
</tbody>
</table>

SWDE-UNIVERSITY. The result is shown in table 4.10. Our framework achieves similar performance with PL+IP+IA in name, website and type. Our framework outperforms PL+IP+IA in phone because the format of the phone is relatively fixed. Our semantic similarity functions is effective to measure the similarity among phone numbers.
To sum up, our framework outperforms baseline algorithms in most of the attributes. Our framework can successfully align the attributes and recognize the data records from many repetitive web components. However, our framework does not perform well when the format of data values are various.

### 4.5.3 Time Efficiency

We analyze the time efficiency of our method. As the time efficiency of data record extraction is important, we adapt the greedy splitting algorithm given in Chapter 3 (Subsection 23) by replacing the optimization function with the semantic similarity function. The time efficiency of our method is given in table 4.11.

<table>
<thead>
<tr>
<th>Vertical</th>
<th>Pages</th>
<th>Total Time (seconds)</th>
<th>Average Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auto</td>
<td>17923</td>
<td>42477</td>
<td>2.37</td>
</tr>
<tr>
<td>Book</td>
<td>20000</td>
<td>51119</td>
<td>2.56</td>
</tr>
<tr>
<td>Camera</td>
<td>5258</td>
<td>12304</td>
<td>2.34</td>
</tr>
<tr>
<td>Job</td>
<td>20000</td>
<td>40801</td>
<td>2.04</td>
</tr>
<tr>
<td>Movie</td>
<td>20000</td>
<td>45118</td>
<td>2.26</td>
</tr>
<tr>
<td>NBA Player</td>
<td>4405</td>
<td>12246</td>
<td>2.78</td>
</tr>
<tr>
<td>Restaurant</td>
<td>20000</td>
<td>49603</td>
<td>2.48</td>
</tr>
<tr>
<td>University</td>
<td>16705</td>
<td>36251</td>
<td>2.17</td>
</tr>
</tbody>
</table>

The average runtime of SAW is from 2.04 seconds to 2.78 seconds. The average runtime is similar in every vertical. The main reasons are: (1) Every web page contains only one item (e.g., one web page contains the information of one book), which is different from the SYNTHIA dataset used in Section 3.5.1 in which every web page contains many repetitive components that need to be split and aligned to extract the template. Separating such a web page into templates and data does not cost much time (c.f. experiment analysis in Section 3.5.4). (2) The efficiency of template clustering and alignment algorithm is similar because the number of template of every vertical is the same. Every vertical has web pages from 10 different web sites. Every web site has one unique tem-
plate. Our template clustering and alignment algorithm thus needs to compare one web page with ten templates to determine which website that the web page belongs to.

## 4.6 Summary

We studied a web page separation problem that aims to extract data records from a set of web pages which are not generated by the same template. We proposed a framework which separates web pages into template code and data records. The framework consists of a semantics-aware web page separation algorithm and a template clustering algorithm. We also conducted experimental study to evaluate the performance of our framework. The result shows that our framework outperforms the baseline algorithms which assume a pre-known clustering of the set of web pages.
5.1 Conclusions

We studied the web page template and data separation problem. The template code and data records are two important aspects of the web page separation problem. With such focus in mind, we studied two sub-problems: web page template and data separation for obtaining (1) high-quality template code, and (2) high-quality data records.

In Chapter 3, for the first sub-problem, we studied a web page separation problem that aims to improve the maintainability of the template code. We utilized an important software metric named the maintainability index to evaluate the maintainability of template code. We proposed a model to analyze the complexity of our web page separation problem and showed that our web page separation problem is NP-hard. We further proposed a heuristic algorithm which processes the DOM tree of a web page recursively in a bottom-up manner with three steps: splitting, folding, and alignment. The splitting step splits siblings in the DOM tree into chunks. The folding step folds a chunk into template code when the alignment algorithm captures differences among siblings. The folding step adds loops to wrap common components while the alignment algorithm adds conditions to capture the additional branches among different siblings. Compared with the previous work [52, 109, 110] which recognized the similar siblings by comparing the tree similarity, our splitting algorithm utilizes the maintainability index of template code to group siblings by similarity. We further proposed a population-based optimization algorithm named dual teaching and learning based optimization (dual-TLBO) for
the splitting step. Dual-TLBO inserts a boundary between two continuous siblings to split them into different chunks. A splitting solution is a list of 0’s and 1’s that record whether there is a boundary between two continuous siblings. Dual-TLBO can find the contribution (increasing or decreasing the MI score) of every element (0’s or 1’s) in a solution. This is achieved by comparing the currently best solution and the currently worst solution pairwise, which helps approach to the optimal solution. For the alignment, we proposed a globally optimal algorithm which can find the best solution within a short running time. We conducted empirical experiments to evaluate the effectiveness of our algorithms. The results showed that our algorithms outperformed baseline algorithms significantly. We studied the effects of parameters in dual-TLBO and the time efficiency based on the size of web pages to provide suggestions on how to adapt our algorithms to various web pages. Results showed that our algorithms were scalable and robust to solve the web page separation for better maintainability.

In Chapter 4 for the second sub-problem, we studied a web page separation problem that aims to extract data records from a set of web pages generated by different unknown templates. Most previous work assumed that the input web pages are generated by the same template. We lifted this assumption and proposed a framework to extract data records from web pages which are not generated by the same template code. Our framework consists of four steps: web page template and data separation, template clustering, template alignment, and data record filtering. The web page template and data separation step separates every web page into template code and data records. The template clustering step clusters web pages by the template code, where web pages in the same cluster are generated by similar template code. The template alignment step captures the differences among template code and constructs a generalized template code that can generate all web pages in the same cluster. The data record filtering step modifies the data records which are incorrectly extracted by the separation step with the generalized template code. We further proposed a template alignment algorithm that utilizes the semantic similarity of data records and the structure of templates to measure the similarity between templates. We conducted an experimental study to evaluate the performance of
our framework. We first evaluate the recognition data records in web pages. Compared with most previous work taking a set of web pages which are generated by the same template, the experimental results showed that our work achieves comparable performance with the state-of-the-art work. Besides, we conducted experiments in recognizing entities. In these experiments, we need to filter out the groups of data records which contain the specific entities. The experimental results show that our framework also achieved comparable performance with the state-of-the-art work.

5.2 Future Work

Our future work focuses on improving separation quality and efficiency.

- We plan to enhance the performance of our framework of extracting data records from a set of web pages. Our current framework relies on the semantic functions, which is vulnerable when there are special symbols, such as “?” and “~”, in the text. In the future, we plan to develop a more robust framework where the performance does not drop dramatically when the semantic similarity functions cannot work well. Besides, the currently extracted data may contain errors, such as attributes of HTML tags. We plan to provide flexible data extraction methods to fix such errors.

- We plan to utilize other features in web page separation. We now use the structure of the sub-DOM tree and the semantics of data records. In the future, we plan to explore other features, e.g., values of attributes, to achieve high-quality separations. Besides, we will study the relationship between the features used and the separation quality.

- As our heuristic algorithm produces local optimal solutions, which are maximizing the MI score of layout trees generated from siblings, we aim to develop a global optimization algorithm. We plan to develop an exact algorithm instead of an approximate algorithm or an approximate algorithm with tighter performance guarantees.

- We also plan to improve the efficiency of our web page separation algorithms. Our web page separation algorithms are slow because dual-TLBO needs to explore
Conclusions and Future Work

many solutions before obtaining the best one. We plan to develop new evolutionary algorithms which approach the best solution faster. Besides, the termination criterion in the current algorithm consists of the generation number and patience number. However, the best solution may be generated before meeting the termination criterion. We will design other termination criterions, for example, to compare the population of solutions between two generations and terminate if there is no new solution generated.

- Our framework is not effective to recognize data records which have similar semantics but different paths. For example, the data values in “\texttt{\langle div\rangle apple\langle /div\rangle}” and “\texttt{\langle div\rangle \langle b\rangle banana\langle /b\rangle\langle /div\rangle}” are both fruit but cannot be aligned because they have different paths. We plan to develop new algorithms to tackle this problem by utilizing the semantics of tag string. For example, “\texttt{\langle b\rangle}” stands for bold and is used to emphasize the data record populated into. The emphasized data records may have similar semantics with other data records which are not emphasized. We may treat “\texttt{\langle b\rangle}” as an attribute (emphasized-or-not) instead of a node.
Bibliography


[45] JOHN WALKER, S. Big data: A revolution that will transform how we live, work, and think, 2014.


[110] Zhai, Y., and Liu, B. Structured data extraction from the web based on partial tree alignment. *IEEE Transactions on Knowledge and Data Engineering* 18, 12 (2006), 1614–1628.

Minerva Access is the Institutional Repository of The University of Melbourne

Author/s:
Zhao, Chenxu

Title:
High-quality lossless web page template and data separation

Date:
2018

Persistent Link:
http://hdl.handle.net/11343/221707

File Description:
Complete Thesis of High-quality lossless web page template and data separation

Terms and Conditions:
Terms and Conditions: Copyright in works deposited in Minerva Access is retained by the copyright owner. The work may not be altered without permission from the copyright owner. Readers may only download, print and save electronic copies of whole works for their own personal non-commercial use. Any use that exceeds these limits requires permission from the copyright owner. Attribution is essential when quoting or paraphrasing from these works.