# Web Page Template and Data Separation for Better Maintainability

Chenxu Zhao, Rui Zhang[⋆], and Jianzhong Qi

School of CIS, The University of Melbourne, Parkville, Australia
chenxuz@student.unimelb.edu.au
{rui.zhang, jianzhong.qi}@unimelb.edu.au

**Abstract.** 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 information extraction. We study this problem with the aim to separate a web page into easily maintainable template code and data records. 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. We perform experiments on real datasets 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 25% in the maintainability measure.

**Keywords:** Web page template extraction · Maintainability index · Dual teaching and learning based optimization

## 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. We study this problem and focus on the maintainability of the template code generated, 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 1 shows an example of the web page separation problem. In particular, Figure 1(a) shows the HTML source of a web page which contains a list of items (cf. the "$\langle li \rangle$" tags). Figure 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 "$\langle for1 \rangle$" tag) to produce a list of "$\langle li \rangle$" items. The attributes and data records to be populated into this list of items are represented by variables "r1" and "r2", the values of which are stored in a data record file as illustrated in Figure 1(c). Note that, in this example HTML source code, the second list item contains an additional "$\langle span \rangle$" tag. This is reflected in the "$\langle if1 \rangle$" tag in the generated template in Figure 1(b).

[⋆] Corresponding author

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

(a) Web page source

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

(b) Template code

```
for1:[
    {r1:java,r2:Java},
    {r1:php,r2:PHP,if1:{}}
]
```

(c) Data records

**Fig. 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 that 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.

SYNTHIA [7] 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, SYNTHIA 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 1(a) is 90, while the total number of characters in the template code and data records in Figures 1(b) and 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 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). 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 paper, 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) [1]. MI is an important software metric to measure the maintainability of software source code. A higher MI score suggests a higher maintainability and is more preferable. Intuitively, a piece of code would have a higher MI score if it is shorter, having few variable and functions, and having few branches (detailed in Section 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 SYNTHIA algorithm to solve our problem. In SYNTHIA, 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 considers 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 with the aim to achieve a high MI score.
– We propose a global alignment algorithm to align siblings to be folded together with the aim to achieve a high MI score.
– We perform a experimental study to evaluate our algorithms in both maximizing the MI score and minimizing the length of the generated template code and data records.

## 2 Related Work

We review closely related work on data record extraction and optimization. MDR [5] uses XPath to identify the text nodes. It presents an XPath matching algorithm to detect template and text nodes with similar XPaths. The partial tree alignment algorithm [11] aligns similar sub-DOM trees. FiVaTech [4] applies alignment algorithms and template detection algorithms on multiple web pages. It defines a new pattern tree named “fixed/variant pattern tree”. These techniques aim to extract the data records only. The web page templates are simply discarded. The original pages cannot be reconstructed from the extraction results because of the loss of template information. TLBO [9] is a population-based optimization algorithm that every solution learns from the current best solution and representative solutions towards achieving the best solution. The number of nodes of a DOM tree may change when deduplicating repetitive sub-DOM trees. Pang et al. [8] present an algorithm for constructing a minimal dominating set on digraphs when the number of nodes is changing.

## 3 Maintainability Index of Template

The *maintainability index* (MI)[1] is a software metric to evaluate the maintainability of programs. We use it to evaluate the quality of a separation. MI is formed by three important software metrics: Halstead Volume (V), Cyclomatic Complexity [6] (G), and Source Lines of Code (SLOC). The function of MI is given as

$MI = 171 - 5.2 \times ln(V) - 0.23 \times (G) - 16.2 \times ln(SLOC).$

We evaluate our work using the MI score and aim to separate the web page into data records and template code that has a high MI score (i.e., better maintainability). We adapt the model given in SYNTHIA [7]. To compute the maintainability index score for a separation, we define the maintainability index of layout trees as summarized in Table 1. Every text node adds one operand and one SLOC. The text node is identified by its value. An element node adds one operator and its starting tag and 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 refers text nodes.

**Table 1.** MI of Layout Trees

| Node Type | Operand | Operator | G | SLOC |
|---|---|---|---|---|
| Text Node | 1 | 0 | 0 | 1 |
| Element Node | 0 | 1 | 0 | 2 |
| For/If Node | 0 | 1 | 1 | 2 |
| Reference Node (attrs) | 1 | 0 | 0 | 0 |
| Reference Node (others) | 1 | 0 | 0 | 1 |

## 4 Problem Definition and Complexity Analysis

### 4.1 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 elements of $S$ (i.e., subsets of siblings) is empty.
2. The union of all elements of $S$ contains all nodes of the DOM tree.
3. The siblings of an element of $S$ 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. We aim to find a variation that has the maximum maintainability index. A formal definition of our problem is as follows.

**Definition 1.** 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.*

### 4.2 Complexity Analysis

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

**Definition 2.** Maintainability based web page separation (decision version) *Given a DOM tree T and the maintainability index function, determine whether there exists a variation whose maintainability index is less than or equal to a given constant c.*

Then, we reduce the *exact cover* problem, which has been proved to be an NP-complete problem [3], 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 maintainability index 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 3.** 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 1.** *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 a number of 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 $c^*$. 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.

## 5 Our Methods

### 5.1 Overall Algorithm Procedure

Our algorithm traverses the DOM tree in a bottom-up manner to generate a layout tree with the aim to maximize the maintainability index 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 5.3) 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 5.2) to capture the differences among the siblings. The splitting algorithm adds iteration nodes, and the alignment algorithm adds condition nodes to the layout tree generated. When the root node of the DOM tree is reached, the algorithm terminates and returns the layout tree generated.

### 5.2 Alignment Algorithm

Our alignment algorithm aligns siblings in the same chunk and captures the differences among the siblings. 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 the following alignment operations: (1) 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. (2)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. (3) 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. (4) 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)$ALIGN[i-1, j-1] + score(A[i], B[j])$, (2) $ALIGN[i-1, j] + score(A[i])$ and (3) $ALIGN[i, j-1] + score(B[j])$. 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. Let $T_a$ be the root of the new layout tree. During tracing back, we add children to $T_a$ to generate an aligned layout tree. If $ALIGN[i, j] = ALIGN[i-1, j+1] + score(A[i], B[j])$, $A[i]$ and $B[j]$ are matched, we align them and add the generated layout tree to $T_a$. If $ALIGN[i, j] = A[i-1, j] + score(A[i])$, 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 $ALIGN[i, j] = A[i, j-1] + score(B[j])$, 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 data records are also aligned to adapt to the changes of the layout tree. The alignment algorithm returns $T_c$.

**Table 2.** Score

| | | $A_2$ | $B_2$ | $D_2$ | $F_2$ |
|---|---|---|---|---|---|
| | 0 | -2 | -5 | -3 | -4 |
| $E_1$ | -4 | -7 | -3 | 2 | 1 |
| $A_1$ | -1 | 6 | -1 | -3 | -5 |
| $C_1$ | -6 | -2 | -1 | -1 | -3 |
| $D_1$ | -4 | -1 | -1 | 5 | -2 |
| $F_1$ | -5 | -5 | -2 | -4 | 6 |

**Table 3.** Alignment

| | | $A_2$ | $B_2$ | $D_2$ | $F_2$ |
|---|---|---|---|---|---|
| | 0 | ← -2 | ← -7 | ← -10 | ← -14 |
| $E_1$ | ↑ -4 | ←↑ -6 | ↖ -5 | ↖ -5 | ←↖ -9 |
| $A_1$ | ↑ -5 | ↖ 2 | ← -3 | ←↑ -6 | ←↖↑ -10 |
| $C_1$ | ↑ -11 | ↑ -4 | ↖ 1 | ← -2 | ← -6 |
| $D_1$ | ↑ -15 | ↑ -8 | ↑ -3 | ↖ 6 | ← 2 |
| $F_1$ | ↑ -20 | ↑ -13 | ↑ -8 | ↑ 1 | ↖ 12 |

To help understand our alignment algorithm, we give a running example. In Tables 2 and 3, 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 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 $score(C_1, B_2)$ is smaller than 0, aligning $C_1$ and $B_2$ with null gets a lower score than $score(C_1, B_2)$.

## 5.3 Dual Teaching and Learning Based Optimization Algorithm

We model the problem of splitting siblings into chunks as a problem of finding boundaries to separate 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). We design a population-

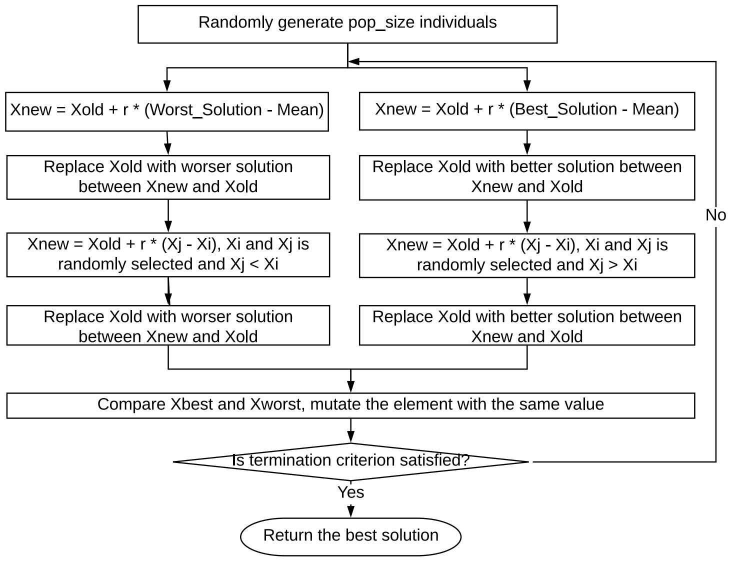
Randomly generate pop_size individuals
Xnew = Xold + r * (Worst_Solution - Mean)
Xnew = Xold + r * (Best_Solution - Mean)
Replace Xold with worser solution between Xnew and Xold
Replace Xold with better solution between Xnew and Xold
Xnew = Xold + r * (Xj - Xi), Xi and Xj is randomly selected and Xj < Xi
Xnew = Xold + r * (Xj - Xi), Xi and Xj is randomly selected and Xj > Xi
Replace Xold with worser solution between Xnew and Xold
Replace Xold with better solution between Xnew and Xold
Compare Xbest and Xworst, mutate the element with the same value
Is termination criterion satisfied?
No
Yes
Return the best solution

**Fig. 2.** Workflow of dual-TLBO

based splitting algorithm which is named *dual teaching and learning based optimization* (dual-TLBO). Figure 2 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 $X_{new} = X_{old} + r(Teacher - Mean)$, 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 $X_{new}$ represents a chunking that yields a higher MI score, we replace $X_{old}$ by $X_{new}$. Otherwise, $X_{old}$ is kept and $X_{new}$ is discarded. Then, it comes to the student phase. We randomly select two individuals $X_i$ and $X_j$ and for every individual $X_{old}$, we create a new individual $X_{new} = X_{old} + r(X_i - X_j)$. Similar to the teacher phase, we compare $X_{old}$ with $X_{new}$ 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.

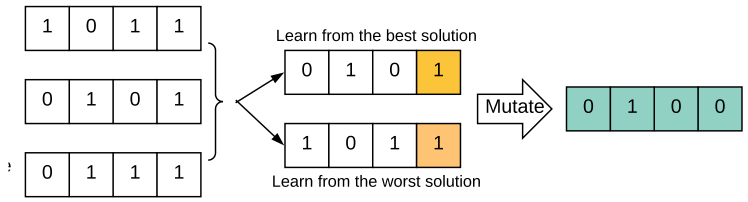

**Fig. 3.** Running Example of dual-TLBO

To help understand the dual-TLBO algorithm, we give a running example next. We assume 5 siblings. The workflow is shown in Figure 3. 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, 1/3, 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 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 maintainability index 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].

## 6 Experiments

We evaluate the empirical performance of our algorithms on real datasets and report the results in this section. We show that our algorithms can produce separations with high 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

SYNTHIA algorithm that is designed to produce separations with small sizes.

We use two real data sets. The first dataset is from the SYNTHIA paper [7]. We denote the dataset by SYNTHIA-DATA. SYNTHIA-DATA contains 200 web pages from 40 websites (5 web pages per website). We use it to compare the performance of our algorithms with the SYNTHIA algorithm. As the code of SYNTHIA is not available, we do our best to implement the algorithm following the SYNTHIA paper [7] for our experiments. The result is shown in Table 4.

**Table 4.** SYNTHIA-DATA

| | MI | CR | time(s) |
|---|---|---|---|
| Original page | -3.46 (0%) | 100% | - |
| SYNTHIA | -3.06 (+0.7%) | 63.34%[1] | 310 |
| MAX-MI-LCS | 6.97 (+19.8%) | 66.31% | 315 |
| MI-Dual-TLBO | *9.38 (+ 24.4%)* | 67.27% | 5943 |

In the table, MI denotes the maintainability index score. CR denotes the compression ratio, i.e., the output file size over the original file size, and time denotes the algorithm response time. We also report CR because SYNTHIA is designed to achieve a small CR value. We report the performance of five algorithms, SYNTHIA is our implementation of the SYNTHIA algorithm [7]; MAX-MI-LCS adds the alignment algorithm described in Section 5.2; MI-Dual-TLBO is our proposed algorithm as described in Section 5.3.

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. 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 and MAX-MI-LCS can be used instead of SYNTHIA as they are as fast as SYNTHIA while obtaining larger MI scores.

**Table 5.** TBDW

| method | dataset | MI | CR | time(s) |
|---|---|---|---|---|
| Original page | TBDW | 17.80 (0%) | 100% | - |
| SYNTHIA | TBDW | 23.02 (+13.45%) | 74.21% | 379 |
| MI-Dual-TLBO | TBDW | *27.50 (+25.0%)* | 75.13% | 2944 |

**Table 6.** UW-CAN

| method | dataset | MI | CR | time(s) |
|---|---|---|---|---|
| Original page | UW-CAN | 41.69 (0%) | 100% | - |
| SYNTHIA | UW-CAN | 45.04 (+6.1%) | 81.16% | 127 |
| MI-Dual-TLBO | UW-CAN | *48.30 (+12.1%)* | 83.92% | 1610 |

We also test our algorithm on other datasets. The result is shown in Table 5 and 6. TBDW [10] is a data set that contains 253 web pages from 51 websites. UW-CAN [2] is a data set that contains 314 web pages from the University of Waterloo and other Canadian

[1] The compression ratio reported in the original SYNTHIA paper is 62.8%.

websites. The result is shown in Table 6. Our proposed algorithm again outperforms SYNTHIA in terms of the MI score, and the advantage is 25.0% and 12.1% on these two datasets, respectively.

## 7 Conclusions

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 proposed a dual-teaching and learning based optimization algorithm to detect siblings generated by the same template. The experimental results show that our algorithms outperform state-of-the-art web page separation techniques. 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. In the future, we plan to work on extracting template code on the website level and data record extraction.

## 8 Acknowledgment

This work is supported by Australian Research Council (ARC) Future Fellowships Project FT120100832 and Discovery Project DP180102050.

## References

1. Counsell, S., Liu, X., Eldh, S., Tonelli, R., Marchesi, M., Concas, G., Murgia, A.: Re-visiting the'maintainability index'metric from an object-oriented perspective. In: SEAA. pp. 84–87 (2015)
2. Hammouda, K.M., Kamel, M.S.: Phrase-based document similarity based on an index graph model. In: ICDM. pp. 203–210 (2002)
3. Karp, R.M.: Reducibility among combinatorial problems. In: Complexity of computer computations, pp. 85–103. Springer (1972)
4. Kayed, M., Chang, C.H.: Fivatech: Page-level web data extraction from template pages. IEEE Transactions on Knowledge and Data Engineering **22**(2), 249–263 (2010)
5. Liu, B., Grossman, R., Zhai, Y.: Mining data records in web pages. In: KDD. pp. 601–606 (2003)
6. McCabe, T.J.: A complexity measure. IEEE Transactions on Software Engineering (4), 308–320 (1976)
7. Omari, A., Kimelfeld, B., Yahav, E., Shoham, S.: Lossless separation of web pages into layout code and data. In: KDD. pp. 1805–1814 (2016)
8. Pang, C., Zhang, R., Zhang, Q., Wang, J.: Dominating sets in directed graphs. Information Sciences **180**(19), 3647–3652 (2010)
9. Rao, R.V., Savsani, V.J., Vakharia, D.: Teaching–learning-based optimization: a novel method for constrained mechanical design optimization problems. Computer-Aided Design **43**(3), 303–315 (2011)
10. Yamada, Y., Craswell, N., Nakatoh, T., Hirokawa, S.: Testbed for information extraction from deep web. In: WWW. pp. 346–347 (2004)
11. Zhai, Y., Liu, B.: Web data extraction based on partial tree alignment. In: WWW. pp. 76–85 (2005)