BACHELOR THESIS
COMPUTER SCIENCE

RADBOUD UNIVERSITY

Visualizing differences between HTML documents

Author:
Aucke Bos
s4591496

First supervisor/assessor:
dr. F (Freek) Wiedijk
freek@cs.ru.nl

Second supervisor:
Ivo Ladage-van Doorn
Ivo.Ladage-vanDoorn@gxsoftware.com

Second assessor:
dr. J.C. (Jurriaan) Rot
jrot@cs.ru.nl

December 11, 2018

Abstract

This thesis concerns itself with the problem of comparing two HTML documents. The algorithm used to solve this problem is based on a paper of Chawathe et al. We have created a library called HDiff, which implements this algorithm in Java. It is also based on an existing library called XMLDiff, which is written in Python. The purpose of HDiff is to support GX Software in the process of releasing updates of their software. It uses the tree structure of HTML to compare its input.

Contents

Chapter 1

Introduction

I decided to do an internship at a company in Nijmegen, rather than solely doing research at the university and writing a thesis about it. The advantage is that I would work on a project of which the result would actually be used by someone or something in the company. I ended up with an assignment at GX Software[1].

1.1 The assignment

The assignment is quite clear, compact, and defined in a few phrases: We need a tool that automatically shows differences between two versions of a HTML page. When a new version of a website is released, we want to be sure that the new version of each page looks exactly how it is supposed to. At the moment we check that manually, by checking every page of the website. The tool should compute the differences, and show them in a user friendly way.

1.2 How to achieve the goal

When looking around at the internet, it becomes clear that this problem is not "new". Many pieces of software try to achieve quite the same thing. Some are focused on showing differences in text only, others compute differences in XML or HTML. The assignment therefore consists of combining and editing some pieces of existing software, and adjust them such that they fit my case.

The remainder of the paper is structured as follows: Chapter 2 defines the question to be answered by this study. Chapter 3 defines some theory applicable to the domain. It introduces several ideas to (a) compare HTML documents, and (b) visualize differences between them. Two possible ways

[1] `https://www.gxsoftware.com/`

to solve our problem are defined in Chapter 4. Chapter 5 describes two libraries related to our problem. Thereafter, in Chapter 6, we define a general outline of how our algorithm works. An already existing algorithm by Chawathe et al, which follows this outline, is presented in Chapter 7 [11]. Chapter 8 then discusses an implementation of this algorithm called XMLDiff, which is written in Python. Our own implementation, HDiff, is written in Java and is based on the algorithm of Chapter 7 and XMLDiff. The differences between HDiff, XMLDiff, and the algorithm of Chawathe et al are described in Chapter 9. Chapter 10 describes the drawn conclusions. It also addresses the work that still has to be done before GX can use HDiff in their systems.

Chapter 2

Research question

The research question is clearly a result of the description of the assignment as given in Section 1.1. We have chosen to try to achieve this goal,comparing HTML documents, by comparing their HTML-code. This results in our research question:

Is it possible to effectively compute and show differences between two HTML documents by comparing their source code?

We stress the meaning of source code here. We *do not* mean the code of the programming language which generates the HTML, but rather the HTML of which the documents consist.

This HTML code is often manipulated by means of JavaScript or other pieces of code. The documents on which we perform our algorithm, are the ones that are rendered by a browser. Such that all JavaScript is correctly executed before we compare the result. This way, we are sure that we compute differences on the exact documents that are shown to the end user.

Please note that this paper is focused on designing and explaining a piece of created software. Because the focus is on practical usage, it does not concern itself with proofs of our algorithm regarding correctness. We have not extensively tested the software, as it still needs adjustments depending on the case it is used for. The algorithm of Chawathe et al is focused on correct an complete output. HDiff is based on that algorithm, but the adaptions are focused on applicability rather than on completeness and correctness.

For some cases the differences in css are more important, while other cases focus more on text differences. For different cases one might want to add different adjustments for efficiency, but also for output format. The case namely also defines what the best format of the output looks like, such that the most important differences are presented in the most user friendly way.

Chapter 3

Theory

Achieving our goal naturally consists of 2 steps. The first step is to calculate the differences between the two files, and the second step is to present the result to the user.

3.1 Ways to compare two HTML documents

Calculating the differences of two HTML objects can be done in several ways.

3.1.1 Using text-based comparison tools

This is probably the least complex way to compare two files. In this case, one does not take into account the tree-structure of HTML: you interpret it as if it were just plain text. For simple HTML files, this might very well work just fine.

However, when pages get more complex, and use elements like `<table>`, `<ul>`, and nested `<div>`'s, chances are high that things will break. If you would still use text-based comparison tools, the merged output file might have invalid HTML syntax. In that case, the HTML cannot be rendered, and thus we cannot show rendered HTML to the user. We do not want to bother the user with the hassle of dealing with HTML code, so this is not an option.

We can however, use a text-based comparison algorithm as *a part of* our complete algorithm. When one knows that two nodes are each others match in both trees, one can compare their inner text using text-based comparison. It is then possible to show specific changes to the user. Examples are the replacements of single words. When you would not use text-based comparison, the best one can do is mark the element containing the text as "changed". The result is that a whole block of text is marked as changed, as soon as a single change in text occurs. This is clearly not desirable. In item

3 of Section 9.2 we discuss the way we've implemented this comparison in HDiff.

3.1.2 Using image-based comparison tools

Another way of looking at this problem is to not take the source code into account at all. Tools exist that can show differences between images. One can take snapshots of (parts of) the two HTML documents, and compare them using those tools. This will probably suffice when the two pages do not differ too much. However, when changes in alignment occur, this method might be inferior to the alternatives, as it is not capable of structurally showing the differences. Small differences can then result in the tool interpreting the files as being completely different from each other.

We have therefore decided to not use this approach in HDiff.

3.1.3 Using the tree structure of HTML

With this method, you take into account the way HTML is structured. The idea is that you build the tree for both of the files. Because HTML "in the wild" is often malformed, this step will need a fixing algorithm. Malformed HTML will render in browser, but it cannot be parsed, and thus a tree cannot be created.

The reason that browsers are able to render malformed HTML, is that the browser actually uses a fixing tool itself. We will therefore simulate a browser in our library. This way, we achieve two goals. The first goal is that the fixing is automatically done for us. The browser will always return well-formed HTML. The second goal relates to the fact that we also need to execute JavaScript, which often manipulates the HTML content. The browser will also do this for us, and then returns the resulting HTML.

When the HTML is converted to a well formed tree, an algorithm needs to be built that can compare the two trees. It should be capable of recognizing which nodes in tree 1 correspond to which nodes in tree 2. This task becomes more complicated when nodes are moved not only inside their parent, but also between parents.

When all nodes are matched, the differences between two corresponding nodes should be computed and saved.

It is possible that a changed node is not correctly matched to its new version in tree 2. For example: tree 1 contains a node:

```
1 <div id="parent">
2     <span>Child 1</span>
3     <span>Child 2</span>
4 </div>
```

And in tree 2, the children are swapped, and the tag of 'Child 1' is changed to :

```
<div id="parent">
    <span>Child 2</span>
    <div>Child 1</div>
</div>
```

Ideally, the algorithm would recognize the swap, and present it to the user as being a swap of two elements, and the rename of a tag. But when the algorithm is not sophisticated enough, it might be unable to match Child 1 in tree 1 to Child 1 in tree 2. It will probably mark them as deleted and inserted nodes. This behaviour is not devastating, but it is not desired either. Tricks can be thought of to fix this issue. One might for example enforce each child in each tree to have an "id" attribute with a unique value. The algorithm can then use these identifiers to match nodes between trees. This would however drastically lower the applicability of the software, as HTML found online almost never has unique identifiers for all elements.

For our implementation, we have decided to follow this approach, while not assuming unique identifiers.

3.2 Comparative designs

This section lists three categories to visualize differences between two complex items: Juxtaposition, Superposition, and Explicit encoding [10].

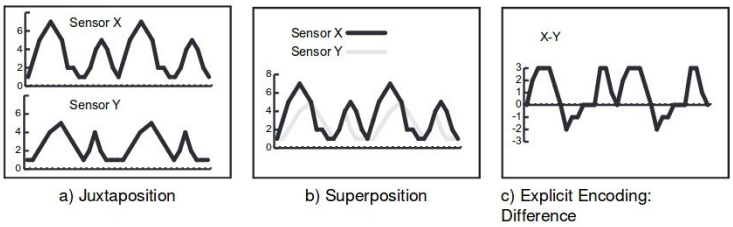
- Using Juxtaposition, one places two objects visually next to each other in space or time. This system relies on the viewers memory.
- Superposition places the objects on top of each other. One overlays the two, and presents them in one place.
- The usage of explicit encoding requires the computation of the differences between two objects. After the computation, only visual encoding of the relationship is shown.

Take a look at Figure 3.1 (which is taken out of Figure 1 of [10]) for a simple visual example of the three representations.

When comparing two objects, one can either use only one of the building blocks mentioned above, or a combination of them. The paper [10] lists that six variants are most common, including the three basic designs and some combinations.

As visualized in Figure 9.1, our implementation uses the hybrid of Superposition and Explicit encoding as explained below in Section 3.2.6. It creates one output file which contains all data of both input files. The differences between them are highlighted by the use of CSS and JavaScript.

Figure 3.1: Visual example of three ways to represent differences



3.2.1 Juxtaposition

Juxtaposition usually occurs in space, rather than time. It is usually easy to design, as it does not need a whole different representation of the objects. The challenge of juxtaposition is to aid the viewer with noticing the relationships. It relies heavily on the human skill to see patterns in similar objects.

3.2.2 Superposition

A superposition design is called an overlay design. It visualizes the objects to be compared in one space. It often requires the modification of one or both of the objects, such that they fit more easily in one picture. Superposition is most usable when the objects to be compared are similar enough that they, and their differences, can be displayed in one space.

3.2.3 Explicit encoding

The main difference between explicit encoding and the first two categories, is that it requires that the relationship between the objects being compared is known. This spares the viewer from needing to find the differences himself, but has as a downside that the relationship between the objects has to be known beforehand.

With this category, a new object is created and shown; the original objects are thus not shown at all. The new object might be of the same form as the initial objects, or it might take another form of representation.

3.2.4 Juxtaposition combined with superposition

Using this hybrid, objects are being shown both in separate spaces, as well as in the same space. As this is quite contradictory, most systems implementing this hybrid actually display both methods separately from each other, which makes it not really a hybrid.

3.2.5 Juxtaposition combined with explicit encoding

Using this technique, both all objects to be compared and their relationships are shown. On the one hand, the juxtaposition aids the viewer by giving context. On the other hand the explicit encoding aids the user with recognizing the relations.

3.2.6 Superposition combined with explicit encoding

This combination can be quite hard to visualize. One needs to show several objects in one space, and at the same time highlight the relationships between them. The highlighting can, however, assist in creating some order in the chaos when superimposing two objects.

Chapter 4

How to handle the assignment

When searching online for existing and usable tools, it becomes clear the result can be achieved in either of two ways.

Chapter 5 describes two suitable comprehensive compassion tools we have investigated. We concluded that understanding such a tool is very hard when the only documentation we have is the documentation in the code. In Chapter 7 and further we discuss the approach that lead us to our end result.

4.1 Using comprehensive tools

This way, we use the most comprehensive tool we can find. This tool should then be built specifically for comparing HTML documents, and visualizing the result. If such a tool can be found, this is the best way to achieve the goal with the least effort.

It is very unlikely that, if such a tool is found, it does *exactly* what we need, and how we need it. It would still need some adjustments.

A tool that purifies HTML, calculates its differences, *and* shows the result is quite complicated, while it still needs adjustments for our specific case.

4.2 Split into subtasks

The alternative is to split the task into subtasks: search online for tools for these tasks, and combine the result. The advantage is that an appropriate tool can be found for each small part of the main task. The result might be more adjusted to the needs than when we would use the first approach.

Chapter 5

Comprehensive comparing tools

We have investigated two comprehensive tools we have found online.

5.1 DaisyDiff

DaisyDiff [8] is the first tool we found that does exactly what we need. The following two sentences on its website describe its working: "*Daisy Diff is a Java library that diffs (compares) HTML files. It highlights added and removed words and annotates changes to the styling.*".

The output of comparing two versions[1,2] of the main page of "news.bbc.co.uk". is shown in Figure 5.1.

Notice the following:

- Some of the css imports are missing, such that the layout is not fully correct.
- Extra css is used to highlight what is changed. The visualization is very clear.
- JavaScript is used to show tooltips when the user clicks on an edited element. This is an effective way to give the user extra information when desired.
- The library has a hard time dealing with images. It marks every image as "removed" and "added", even when its not changed.
- When we look at the parts "Last updated" and the itemize below the text of "Polish archbishop quits amid row", we see that it deals

[1] `http://web.archive.org/web/20070107145418/http://news.bbc.co.uk/`
[2] `http://web.archive.org/web/20070107182640/http://news.bbc.co.uk/`

Figure 5.1: Example output of DaisyDiff

with text changes quite well. It does not mark the whole node as "removed" and "added", but shows the differences inside the text.

We found out that several bugs exist in this library. We fixed some of them, but fearing that new bugs would keep appearing, we decided to look for alternative software. This library has not been updated for a while, and little documentation can be found online.

We have therefore chosen not to use this library directly in our own implementation.

5.2 HtmlDiff

HtmlDiff [7] is written in Python. HDiff is in Java, but we have used parts of HtmlDiff as a basis for our own library.

A website[3] found online lists several tools for comparing HTML documents. It says the following regarding this library: "Quite slow for large files, but handles radical changes very well". We have taken a look at this software, to see which parts we can use.

[3] `https://www.w3.org/wiki/HtmlDiff`

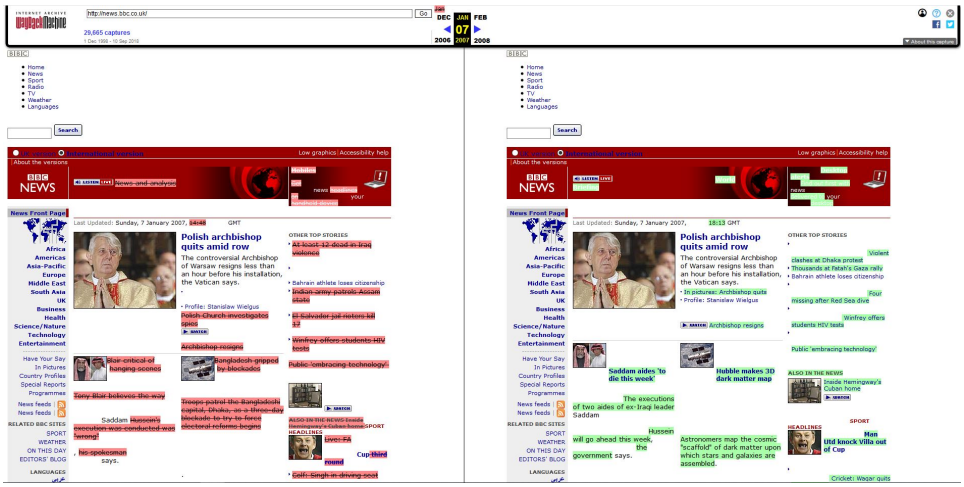
Figure 5.2: Example output of HtmlDiff

We ran it with the same input as we did with DaisyDiff. The output is shown in Figure 5.2.

Notice the following:

- Images are not highlighted, but just replaced. All images contain the source of the second page, and it is not visualized when they differ from the first page. The problem here lies within the core of the library, and the way it parses its input. This is explained further in this Section.
- The differences are shown quite similar as with DaisyDiff.
- The changes in text look even more "subtle" than with DaisyDiff. For example: in the text of item with old title "Blair critical of hanging scenes", the words "Saddam" and "says" are not highlighted to be changed. This is because they occur in both versions at a different location. This library looks at the changes inside the inner text of

Figure 5.3: Example output of HtmlDiff in a side-by-side view



the elements, instead of just marking an element as "changed" as a whole. It uses text-based comparison as described in Section 3.1.1.

The source code of this library is much less complicated than DaisyDiff.

The first thing we notice is that this library has an extra function, which is quite interesting. It is able to show the differences in a side-by-side view, instead of merging them in one file. This gives us the Juxtaposition (3.2.1) method, instead of Superposition (3.2.2). The way the output looks like using this visualization is shown in Figure 5.3.

The algorithm is divided into three main steps. We enumerate and explain them below.

5.2.1 Parsing the input

The algorithm gets files as input. It reads the contents of each file to a string, and then runs $split\_html(s)$ on both of them. This function splits the string into a list of elements in roughly the following way:

```
1  pointer = 0
2  result = []
3  while pointer < s.length
4      if character at pointer is the start of a tag//Either the
5      // start or end of a block
6          result.append(s until the '>')
7      else
8          result.append(one word)
9      pointer += length of added part
10 return result
```

Determining what "a word" is, is done via the following regular expression:

```
([^ \n\r\t,.&;/#=<>()-]+|(?:[ \n\r\t]| )+|[,.&;/#=<>()-])
```

So it adds every word and special character as a new element.

It never matches opening and closing tags of elements. So when an element `<div id="child1">` is added, it is never registered if and when that element is closed by a `</div>`. This way of splitting the document does not really feel natural. It shows that the algorithm does not use the tree structure of HTML, which makes it really hard to match nodes between two trees. It also results in a list of elements with several types:

- Opening tags, including attributes. Examples are `<script type="text/javascript">`, `<div>`.
- Closing tags. Examples are `</div>`, `</title>`.
- Text inside elements. Examples are `\n\t`, `' '`.

This raises the question how the program deals with this differences in attributes in contrast to how it deals with changes in text.

5.2.2 Calculate differences

This is obviously the core of the library. It uses part of another library, called difflib [1]. In fact, the HTMLmatcher used in this part extends the SequenceMatcher of difflib. The SequenceMatcher is unaware of the format of its input: it sees it as a list of plain text elements. This is the reason that the first step is essential, without it, this step would break the HTML structure.
It generates so-called opcodes, which are then used to created a merged document of both inputs. These opcodes are generated after the calculation of matching sequences of the input. We explain both parts:

5.2.2.1 Get matching sequences

This part uses a function $get\_matching\_blocks(a, b)$. It returns a list of triples (i, j, n), indicating that $a[i : i + n] == b[j : j + n]$. It finds the longest match using the gestalt approach [9]. In short this works by finding the first longest common block, adding it to the list of matching blocks, and recursively applying the same function to the left and the right side of the block.

A function $is\_junk(s)$ is added. This function defines some type of characters that may never be inside a longest match, but are allowed at one or both ends of a matching block. HtmlDiff defines some special characters

as junk, and a list of stop words like "a", "how" and "it". $is\_junk(s)$ is not used when ran with the `--accurate-mode` option, to improve the quality of the result, while decreasing efficiency.

5.2.2.2 Get opcodes

The documentation of the library explains how the function $get\_opcodes(a, b)$ works. We discuss the format of the output, and how the output is computed.

5.2.2.2.1 Output An opcode is a 5-tuple of the form $(tag, i_1, i_2, j_1, j_2)$, and represents one step in the task of rewriting input a to input b. tag indicates the operation that has to be performed on $a[i_1 : i_2]$ and $b[j_1 : j_2]$.

For two neighbouring tuples in the list of opcodes, i_2 in tuple 1 is equal to i_1 in tuple 2; likewise for j_2 and j_1. This ensures that the opcodes describe a complete list of operations that have to be performed.

The four possibilities of the value of the tag explained in the documentation complete the understanding:

`'replace'`: $a[i_1 : i_2]$ `should be replaced by` $b[j_1 : j_2]$.
`'delete'` : $a[i_1 : i_2]$ `should be deleted.`
`Note that` $j_1 == j_2$ `in this case.`
`'insert'` : $b[j_1 : j_2]$ `should be inserted at` $a[i_1 : i_1]$.
`Note that` $i_1 == i_2$ `in this case.`
`'equal'` : $a[i_1 : i_2] == b[j_1 : j_2]$.

5.2.2.2.2 Computation The function starts at the beginning of both strings, and loops over the calculated matching blocks. It keeps track of the current position i at input a and j at input b. On every iteration, it creates two opcodes.

1. It firstly creates an opcode (tag, i, a_i, j, b_j). i and j are, as mentioned, the current positions, and a_i and b_j are the indices for a and b in the matching block that is currently being evaluated. tag is defined by:

   ```
   if i < a_i and j < b_j:
       tag = 'replace'
   elif i < a_i:
       tag = 'delete'
   elif j < b_j:
       tag = 'insert'
   ```

   If the current positions are both smaller than the start of the next matching block, $a[i : i + a_i] \neq b[j : j + b_j]$, and thus the first one should be replaced by the second one.

If only $i < a_i$, and thus $j \geq b_j$, $a[i : a_i]$ should be deleted to get both pointers at the start of the matching block.

If $j < b_i$, and $i \geq a_i$, $b[j : b_j]$ has to be inserted at $a[i]$.

2. It then moves i and j to the end of the matching blocks, and creates opcode $('equal', a_i, i, b_j, j)$. This defines that $a[a_i : i] == b[b_j : j]$, which is the matching block. Now the pointers are at the end of this matching block, and the next one can be evaluated.

5.2.3 Creating the output

This is the last part of the library. It now knows the difference between a and b, which is defined as a list of opcodes. It naturally loops over the opcodes, and for each opcode $(tag, i_1, i_2, j_1, j_2)$ writes, depending on the value of tag, the following to an initially empty output file:

- tag == "equal"
  Write $a[i_1 : i_2]$ to the output.

- tag == "delete"
  For each item in $a[i_1 : i_2]$, do the following:
  If the item starts with "<", in other words, the item is not inner text but an element tag, skip this element[4]. Else, write the complete inner text wrapped in a `<span class="delete">` element to the output.

- tag == "insert"
  For each item in $b[j_1 : j_2]$. do the following:
  If the item starts with "<", write the element tag to the output. Else, write the complete inner text wrapped in a `<span class="insert">` element to the output.

- tag == "replace"
  If each element in $a[i_1 : i_2]$ and $b[j_1 : j_2]$ starts with "<", just write $b[j_1 : j_2]$ to the output. Else perform the delete of $a[i_1 : j_2]$ and the insert of $b[j_1 : j_2]$ as shown above.

The algorithm then finishes up by adding a simple CSS stylesheet for the "insert" and "delete" classes.

It is now clear what goes wrong in this library: It does not show differences *inside* element tags. This results in the fact the it only shows inserts, deletions, and changes of inner text.

The biggest problem here is that the program does not use the tree structure of HTML. So when it notices a tag to be different, it could mark it as

[4]Notice that it now becomes clear why changed images are not being highlighted. The algorithm *only* shows differences in inner text, not in element tags.

"added", and mark the old one as "deleted", but nothing more. This results in an output file that shows the fact that an element has changed attributes, but the old element with its inner HTML cannot be shown anymore. The following example clarifies why:

The first input contains the following snippet of HTML.

```
1       ...
2       <div style="color:red">
3           Inner HTML
4       </div>
5       ...
```

The file is changed, such that the following snippet is now present in that place.

```
1       ...
2       <div style="color:blue">
3           Inner HTML
4       </div>
5       ...
```

The opcodes will say that line 2 in the first input is changed to line 2 in the second input, and lines 3 and 4 are unchanged. It will then perform $text\_delete(t)$ on line 2 of input 1 and $text\_insert(t)$ on line 2 of input 2. Initially, it would have then written nothing to the output in the first function, and it would have written the exact line 2 of input 2 to the output in the second function. This would have shown no differences at all, as it just skips tags.

The way we can solve this, is to created an element with class "deleted", without its original content, and immediately close it. Thereafter we add the new element with class "added", with its new content. We cannot add the new content to the old element, as there is no way to be sure that that these elements are even the same elements in both files. And we surely do not want to add the wrong content to the "deleted" element. Input 2 could just as well be as follows.

```
...
<div style="color:blue">
    Inner HTML
</div>
<div style="color:red">
    Inner HTML
</div>
...
```

Which suggests that the first element of both files should not even be seen as a match at all.

This problem cannot be solved without changing the core of the algorithm. The problem lies within the first of the three parts: the way the input is split.

HDiff splits the input using its tree structure, and uses the approach of HtmlDiff to compare inner texts.

Chapter 6

Outline of a possible algorithm

Now that we have seen two implementations of showing differences in HTML, we have got some insight in how an end solution should look like.

If we split the input while keeping its HTML structure, we get a big advantage over the approach of HtmlDiff: we can use an algorithm that matches (sub)trees between the files. If that is done correctly, we are sure whether element *a* in input 1 is a match to element *b* in input 2. We are then able to show the user which parts of which elements have been changed in which way, and which elements have been added or deleted as a whole.

The main requirements of our solution are:

- The solution should deal with both inner HTML data and attribute changes.
- To show differences in a user friendly way, the solution should be aware of the tree structure of HTML.
- The solution should be able to match nodes accurately and in a logical way.

Chapter 7

A working algorithm

After defining the general outline of the needed algorithm in Chapter 6, it is a little clearer what sort of algorithm we need. We need an algorithm that computes a matching of two tree structured data inputs. We found a paper by Chawathe et al [11] that describes an algorithm to detect changes in tree structured data. This section is dedicated to explaining that algorithm extensively, making use of example trees where deemed necessary. The implementation of the algorithm is discussed in Chapters 8 and 9.

7.1 Four characteristics

The introduction of the paper introduces four "key characteristics". We state them here, and define the applicability to our case

- **Nested Information**
  Idea: The algorithm deals with *hierarchical information*. It identifies changes to the nodes themselves, but also to their relationship to the data structure as a whole.
  Applicability: This is exactly what we need. As we have shown in Chapter 6 as a result of the findings in Section 5.2, one of the requirements of our algorithm is that it needs to be aware of the tree structure of our data.

- **Object Identifiers not assumed**
  Idea: Because of this characteristic, the algorithm does not assume the presence of unique identifiers. If there were unique identifiers for all nodes, these could be used to uniquely match nodes between trees. Since these identifiers are assumed not to be present, nodes are matched on their contents instead.
  Applicability: We have shortly mentioned the usage of unique identifiers in Section 3.1.3. If we would have unique identifiers for each element in the first tree, and these identifiers are unchanged in the

second tree, this would highly simplify the matching part of the algorithm. As HTML found online rarely has unique identifiers for every element, we need an algorithm that can also match on other characteristics.

- **Old, New Version Comparison**
  Idea: The algorithm is focused on detecting changes not in two totally different inputs, but rather on two versions of the same data.
  Applicability: In our case, we will apply the algorithm on two versions of the same HTML page, so this is what we need.

- **High Performance**
  Idea: The algorithm uses features common in many applications, for performances sake. It should therefore be more efficient than other solutions, but might sometimes find non-minimal *although still correct* deltas.
  Applicability: The fact that it sometimes finds non-minimal solutions is not really a problem. As long as the output is correct, we can use it. If some changes are shown a little non-intuitive, the output is still usable.

The paper continues by defining some preliminaries, and explains the algorithm in detail. We will follow this outline in the remaining of this Chapter. The differences between HDiff and the paper are enumerated in Sections 8 and 9.

7.2 Preliminaries

The input consists of two trees T_1 and T_2. A tree has a root and descending nodes; each node has a label, a value, and a unique identifier. Identifiers *may be* generated by the algorithm. An element in the first tree might have another id than its match in the second tree, such that the identifiers cannot be used while computing a matching. They are only used to reference nodes: node with id x will be called "node x" having label $l(x)$, value $v(x)$, and parent $p(x)$.

The algorithm firstly needs to find pairs of nodes that correspond to each other in both trees. If two leafs have the exact same value, they *probably* correspond. We say "probably" because this does not only depend on the value of a node, but also on its position in its parent. Because of this, two nodes do not have to be exactly equal for them to be matched. When two nodes have the same position in the same parent, but have differing values, they might still be the best match. So we have to define a function that computes whether two nodes are "equal", which basically means they are similar enough in label, value, children, and position in parent.

A set of matching node tuples is called a matching M, which is always one-to-one. This means that a node can be matched to at most one other node. A partial matching considers some but not all nodes of two trees, whereas total matching takes all nodes into account.

Two trees are isomorphic if their only differences are the identifiers. We want to find a sequence of operations that changes T_1 into T_1', where T_1' is isomorphic to T_2, which we denote as $T_1' \cong T_2$. The partial matching M for T_1 and T_2 can then be extended to a total matching M' between T_1' and T_2, as they are isomorphic.

A *sequence* of operations e is called an edit script E. We call this a sequence and not a set, as the order of the operations matters. The working of the algorithm decides the order of the sequence. When we for example need to insert a new node that has a descending leaf, the node obviously has to be inserted before its leaf.

It is best to rewrite T_1 to T_1' with the least effort possible, as this results in the most intuitive edit script. We therefore define the cost for an edit operation, where the cost of an edit script is the sum of the costs of its edit operations.

We define $v_1, \ldots, v_m$ as children of node u, and call v_i the i^{th} child of u. Recall that a node x has label $l(x)$, value $v(x)$, and parent $p(x)$. There exist four edit operations.

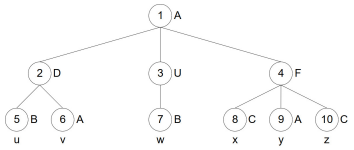
- **Insert**: INS$((x, l, v), y, k)$ means that node x with label l and value v is inserted as the k^{th} child of node y.
- **Delete:** DEL(x) removes *leaf* x from the tree. To delete an inner node, we recursively delete its descending children before deleting the node itself.
- **Update:** UPD(x, val) performs $v(x) = val$.
- **Move:** $MOV(x, y, k)$ sets x to be the k^{th} child of y; x's children move along with x.

Figure 7.1 shows an example for each of the four possible edit actions. The numbers denote the identifiers; the capital letters the labels, and small letters define the values of the leafs. Inner nodes do not have values in this example.

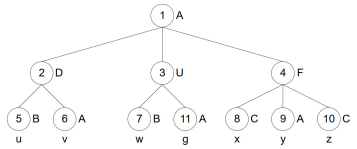
Now an edit script E is a sequence of edit operations e_i. $E = e_1, \ldots, e_m$ takes T_1 to T_{m+1} if there exists $T_2, \ldots, T_m$ such that $T_1 \xrightarrow{e_1} T_2 \xrightarrow{e_2} \ldots \xrightarrow{e_m} T_{m+1}$. An edit script of T_1 with respect to T_2 is said to transform T_1 to T_2: $T_1 \overset{E}{\longmapsto} T_2$ if and only if $T_1 \xrightarrow{E} T_1' \wedge T_1' \cong T_2$.

As mentioned, we define the cost of an edit script to be the sum of the costs of its edit operations. We define $c_D(x)$, $c_I(x)$, $c_U(x)$, $c_M(x)$ respec-

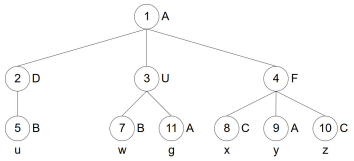
Figure 7.1: Examples of the operations



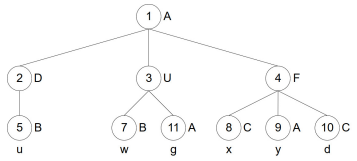
(a) The initial tree T



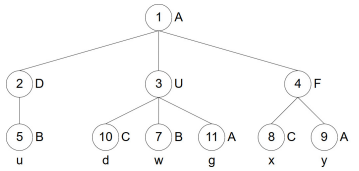
(b) T after $INS((11, \mathrm{A}, \mathrm{g}), 3, 2)$: Insert a node with identifier 11, value A and label g into node r at position 2.



(c) T after $DEL(6)$: Delete node 6



(d) T after $UPD(10, \mathrm{d})$: Update the value of node 10 to d



(e) T after $MOV(10, 3, 1)$: Move node 10 to position 1 in node 3

tively to be the cost of deletion, insertion, update, and move of node x; $c_D(x) = c_I(x) = c_M(x) = 1$.

To calculate $c_U(x)$, we need a $compare(v, v')$ function, that checks how different x's old value v and its new value v' are. The result is a value between zero and two. If v and v' are *similar enough*, the result of *compare* is < 1, else it is ≥ 1. Using this cost model, deleting and inserting a node which value has not changed a lot, is more expensive than just updating its value. The result is that our model prefers updates over deletions/insertions when two nodes are similar. This will give a more intuitive edit script.

7.3 The good matching problem

This part of the algorithm is responsible for creating a *good matching* M. Recall the second characteristic explained in Section 7.1. As a result of this characteristic, we assume keyless data. Therefore we must match nodes on their label and value, and not by their unique identifiers.

7.3.1 Matching criteria

There is obviously more than one way to match two trees. Hence we define some criteria which a matching must satisfy.

M is better than M' if edit script E_M conforming M is cheaper than edit script $E_{M'}$ conforming M'. We try to find the best M for the considered trees T_1 and T_2. The goal of each of the following criteria is to create meaningful matches, or to make the matching algorithm more efficient.

1. **Criterion 1: Do not match dissimilar leafs**
   Recall the $compare()$ function as discussed in Section 7.2. It takes two values and returns the distance between them as a value between 0 and 2. We define a parameter $0 \leq f \leq 1$, which is the maximum distance the values of two nodes may have for them to be matched in M. So for all $(x, y) \in M$, $compare(v(x), v(y)) \leq f$. We also state that $l(x) = l(y)$ holds for all $(x, y) \in M$. Thus two nodes must have equal labels for them to be matched.

2. **Criterion 2: Do not match dissimilar nodes**
   We also do not want to match *inner* nodes are not similar. For an inner node x, we say that x contains y if y is a leaf in the subtree rooted at x. $|x|$ is the total number of leaf nodes that x contains. We now define a set $common(x, y) = \{(u, v) \in M | x \text{ contains } u, \text{ and } y \text{ contains } v\}$; for internal nodes x and y to be matched, $\frac{|common(x,y)|}{max(|x|,|y|)} > t$. t is another parameter, indicating how similar x's and y's descendants must be for x and y to be matched. Its value is defined by $\frac{1}{2} \leq t \leq 1$. So we match inner nodes x and y only if enough of the leafs they contain are matched. For inner nodes x and y it also holds that if $(x, y) \in M$, then $l(x) = l(y)$.

3. **Criterion 3: There is at most one good match for each leaf**
   This criterion is concerned with both the type of input, and the definition of the algorithm. It states that for any leaf $x \in T_1$, there is at most one leaf $y \in T_2$ such that $compare(v(x), v(y)) \leq 1$, and the other way around for all leafs $y \in T_2$. So for all leafs in T_1 or T_2, there is at most one leaf in the other tree that is similar enough for them to be matched.

   This means that the $compare()$ function is defined such that it outputs at most one match for each node. As we will see later, this criterion cannot be true for all types of data that the algorithm is used on.

7.3.2 The algorithm

We define separate $equal()$ functions for leafs and inner nodes. They rely on the $compare()$ function we introduced earlier. Two nodes are thus ought to

be "equal" when they are similar enough, as we have already mentioned in Section 7.2. The definitions of *equal* for leafs and inner nodes respectively are as follows:

$$equal(x,y) = \begin{cases} true \text{ if } l(x) = l(y) \text{ and } compare(v(x), v(y)) \leq f \\ false \text{ otherwise} \end{cases}$$

$$equal(x,y) = \begin{cases} true \text{ if } l(x) = l(y) \text{ and } \frac{|common(x,y)|}{max(|x|,|y|)} > t \\ false \text{ otherwise} \end{cases}$$

These equations show that we must traverse bottom up: children need to be matched before their parent, as the *equal* function of inner nodes relies on the matching of their descending leafs.

With this knowledge, our matching algorithm is straightforward:

```
1 M = []
2 Mark all nodes in T1 and T2 as unmatched
3 for all unmatched nodes x in bottom_up_traverse(T1)
4     for all unmatched nodes y in bottom_up_traverse(T2)
5         if equal(x,y)
6             M.append((x, y))
7             Mark x and y as matched
8             break
```

7.4 Generating the edit script

The problem of generating an edit script is stated as follows: Given a tree T_1, a tree T_2, and a (partial) matching M between their nodes, generate a *minimum cost* edit script E conforming M and transforming T_1 to T_2.

The algorithm generates edit operations and applies each operation to T_1 as soon as one is created. When the algorithm terminates, we have a complete minimum cost edit script, and a complete matching M', which is an extended version of M.

We call a node that is not matched in M and *unmatched* node. A *matched* node has a corresponding *partner* in the other tree. The algorithm is split into five parts, which are explained below.

- **Update**
  In this part, we look for partners $(x,y) \in M$ with differing values, and update the value of x to the value of y:
  For all $(x,y) \in M$ where $v(x) \neq v(y)$: Create $UPD(x, v(y))$.

- **Align**
  This step creates *MOV* operations to move children *inside* their parents until all partners have the same position in their parent. To clarify, let us say we have parents x, y, where $(x, y) \in M$. x has children u and v, which have partners u' and v' being the children of y. When u is to the left of v, but u' is to the right of v', u and v are misaligned, thus we need a *MOV* operation.

  How we efficiently define the set of needed *MOV* operations, will be explained in Section 7.4.1. We use the *Largest Common Subsequence* for this problem.

- **Insert**
  Before this step, we need to be sure that the roots are matched. So when the roots are not already matched in M, we create new roots x and y for T_1 and T_2 respectively. The old roots x' and y' are made the sole child of x and y, and x and y are matched. We now have matched the roots for both trees.

  We look for *unmatched* nodes z in T_2, where $p(z) = y$, and $(x, y) \in M$. We then create a new identifier w, and create $INS((w, l(z), v(z)), x, k)$. Thus we copy z, and insert the new node at position k in the partner of z's parent. We then append (w, z) to M.

  How position k is calculated, is explained in Section 7.4.2; it uses the number of already aligned children in x.

- **Move**
  In this step, we look for partners x and y whose parents are not matched: $(x, y) \in M, p(x) = u, p(y) = v, (u, v) \notin M$. We then create $MOV(x, s, k)$, where $(s, v) \in M$. We are sure that s exists because we have finished the Insert step already. So we move x into the partner of y's parent at position k. k is calculated in the same way as in the Insert step, as explained in Section 7.4.2.

- **Delete**
  This is the last step. At this point, T_1 is almost isomorphic to T_2. The only difference is that T_1 has some unmatched nodes. Thus we traverse T_1 bottom up, and for each unmatched node x, we create and apply $DEL(x)$.

To clarify the working, take a look at Figure 7.2. It shows two trees T_1 and T_2, where T_2 is an altered version of T_1. The nodes are already matched: matching nodes have the same color; white nodes are unmatched. The numbers inside the nodes are *unique identifiers*; the small letters are the *values*; the capital letters are the *labels*. The red arrows indicate the position in the tree on which the associated action has effect. The labels represent

HTML elements:
`<body>`, `<div>`, `<ul>`, `<figure>`, `<a>`, `<p>`, `<li>`, `<img>`, `<span>`.
The associated documents, `example_T_1.html` and `example_T_2.html`, can be found in the Appendix, together with their visual representation. Note the following about these documents:

- HTML attributes are ignored in this example. They are added in the HTML document such that it looks like a normal HTML page, or to force that the nodes are matched as displayed in Figure 7.2.
- Node 8 in `example_T_1.html` has its label changed to `<span>` in Node 20 in `example_T_2.html`. The reason we do that is that the HTML standard prevents us from having a `<li>` within a parent that is not of type `<ul>`. The currently explained version of the algorithm would not allow node 7 and 20 to be matched if they do not have the same label, which is why we set the label of node 20 in T_2 to L.
- Node 3 and node 16 only have one equal child. Because node 3 has 3 children, $\frac{|common(3,16)|}{max(|3|,|16)} = \frac{1}{3}$. As mentioned in Section 7.3.1, $\frac{1}{2} \leq t \leq 1$. So $equal(3,16)$ would be false. In order to still get this matching, we should adjust t to have a value $\leq \frac{1}{3}$. We ignore this fact as the example is focused on clarifying the creation of the edit script, not the creation of the matching.

We perform the algorithm on the trees, showing the current T_1 after each step in Figure 7.3. Confirm that the algorithm works on this example, as $T_1 \cong T_2$ holds after the last step.

Figure 7.2: The matching of T_1 and T_2

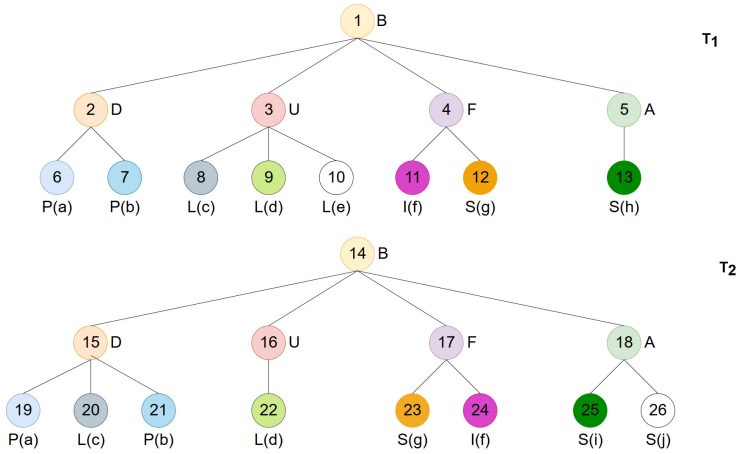
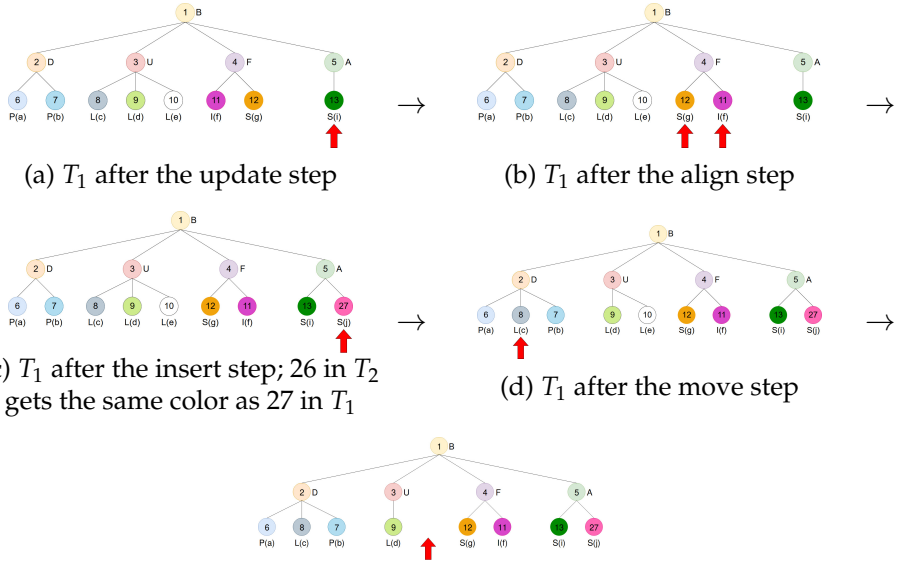


Figure 7.3: An example of edit script algorithm



(a) T_1 after the update step

(b) T_1 after the align step

(c) T_1 after the insert step; 26 in T_2 gets the same color as 27 in T_1

(d) T_1 after the move step

(e) T_1 after the delete step; $T_1 \cong T_2$

7.4.1 Aligning children

In general, there is more than one way to align the children of a node. Thus we have to use an algorithm that aligns all children of x with the least possible edit actions. To get this sequence of actions, we use a solution to the *Longest Common Subsequence* problem [4]. The algorithm takes three inputs: two sequences S_1 and S_2, and a function $equal()$.

If the children of x are misaligned, and $(x, y) \in M$, the sequences are the children of x and the children of y. The *equal* function defines node u and v to be equal when $(u, v) \in M$. The algorithm computes the longest common subsequence of the two input sequences. We then mark all children of x that occur in that sequence to be properly aligned. For all other children, we create and apply edit actions for moving them to the correct position, relatively to the already aligned children. Thereafter we are sure that all children of x are aligned.

7.4.2 Finding the right position

This function is used to find a position at which a node has to be inserted. The input is a node $x \in T_2$. We define $y = p(x) \in T_2$. If x is the first child in y that is yet in order, we immediately return 1. Else, we find the rightmost in-order node $v \in T_2$ that is to the left of x. Now $u \in T_1$ is v's partner. We count the number of elements to the left of u that are in order, and return that number + 2, such that the returned position will be directly to the right

of u. This is the position at which the partner of x is to be inserted.

7.4.3 The algorithm

Having the definition of each step, we define the edit script algorithm:

```
1  E = []
2  M' = M
3  for all nodes x in bread_first_traverse(T2)
4      y = p(x)//y in T2
5      z = partner of y in M'//z in T1
6      if x does not have a partner in M'
7          k = find_position(x)
8          e = INS((w, l(x), v(x)), z, k)//w is a copy of x
9          M'.append((w, x))
10         E.append(e) and apply e to T1
11     else if x is not the root
12         w = partner of x//w in T1
13         if v(w) is not equal to v(x)
14             e = UPD(w, v(x))
15             E.append(e) and apply e to T1
16         v = p(w)//v in T1
17         if (v, y) not in M//The parents of w and x are not matched
18             k = find_position(x)
19             z = partner of y in M'
20             e = MOV(w, z, k)
21             E.append(e) and apply e to T1
22     align_children(w, x)
23 for all unmatched nodes x in post_order_traverse(T1)
24     e = DEL(x)
25     E.append(e) and apply e to T1
```

Chapter 8

XMLDiff

The algorithm explained in the previous Chapter is already implemented in Python, in a library called XMLDiff [2]. As we learned in Section 5.1, understanding a comprehensive library can be quite a challenge. But now we completely understand the algorithm on which it is based; this helps a lot!

We start by explaining how HTML is parsed to a tree. Thereafter, we will explain how both phases of the algorithm are implemented in this library. It will mainly consist of explaining noteworthy differences between the library and the paper. Note that in HDiff, some things differ even more from the paper. The changes regarding HDiff with respect to XMLDiff are discussed in Chapter 9.

8.1 Parsing the input

HTML is parsed by a Python library called "etree". With this parser, elements can have two types of text: Inner text and Tailing text. We explain how these values assigned to which nodes by the use of the following sample code:

```
<div id="1">
    text1
    <p id="2">
        paragraph
    </p>
    text2
</div>
```

Now "text1" is, quite obviously, the inner text of node 1. "paragraph" is the inner text of node 2. The text "text2" is *not* assigned to its parent, but is the tailing text of node 2.

Whenever text occurs in source code, we check the tag that was evaluated just before that. If its a closing tag (`</p>`), the text is assigned to be the tailing text of that element. If its an opening tag (`<div...>`), it will be the inner text of that element.

This way, nodes always have at most 1 value for the inner text and the tailing text. If we would have assigned "text2" to be the second inner text of node 1, we would get elements containing a whole lists of texts.

An HTML node now has four characteristics instead of two described in the paper:

- Tag. This is equal to the label in the paper
- Inner text. This is equal to the value in the paper
- Tailing text.
- Attributes. HTML tags can, and often do, have attributes. Examples are "class", "style", and "type".

8.2 Finding matches

The outline of the library regarding the matching part is quite the same as described in the paper. The main differences can be found in the three criteria discussed in Section 7.3.1.

Remark that the $compare()$ function of this library computes similarity instead of distance. So the variable f is not used as the maximum distance, but rather as the minimum similarity. Its value is just transformed to be $1 - f$.

1. **Criterion 1: Don't match dissimilar leafs**
   The library drops the requirement of $l(x) = l(y)$. This feels very natural. Suppose we have two elements:
   `<div id="child1">This is  child 1</div>` and
   `<span id="child1">This is  child 1</span>`. We obviously want them to be matched. So for HTML, the equality of labels is definitely not a must.

   The requirement of $compare(v(x), v(y)) \leq f$ (or in the case of similarity instead of distance: $> f$) is still used. For the value of a node, its inner text is used.

   The variable f is a user-defined parameter.

2. **Criterion 2: Don't match dissimilar nodes**
   To compute the similarity of two inner nodes x and y, the paper runs the compare function for both inner nodes and the one for leafs. The

reason behind this is probably that inner nodes are as much important as leaf nodes in the "meaning" of an HTML document.

The paper is focused on structures where leafs contain almost all information of the tree, and inner nodes mainly exist to define the structure (like in LaTeXdocuments). In HTML, this is not the case. Not only leaf nodes have characteristics, but inner nodes do as well.

To distinguish the two compare functions, we call the compare function for inner nodes $compare_{inner}$, and the compare function for leafs $compare_{leaf}$. For leaf nodes, only the $compare_{leaf}$ is used to define similarity. For inner nodes, both the $compare_{inner}$ and the $compare_{leaf}$ are used.

$compare_{inner}$ is equal to the one described in the paper, apart from the fact that it is based on *immediate children* instead of *descending leafs*. This again shows the importance of inner nodes over leafs. When comparing two inner nodes, their direct children are a better indication of their similarity than their descending leafs. The latter could exist in a much lower level of the sub tree, and thus not say anything meaningful about the node currently being evaluated.

The variable t is not used. To compute the final similarity of two inner nodes x and y, the average of $compare_{inner}$ and $compare_{leaf}$ is taken. This result is ought to be $> f$. This is probably chosen to drop the differences between inner nodes and leaves even more. When you reason that inner nodes are as important as leafs, it does not really make sense to have different requirements regarding the minimum similarity. This way they indicate that value comparison and child comparison are of equal importance.

3. **Criterion 3: There is at most one good match for each leaf**
   As we've mentioned in Section 7.3.1, this criterion is dependent on the type of input. As it is very likely that more than one node in T_2 can be matched to node x in T_1, this criterion does not hold for HTML. The simple and realistic example of several `</br>`'s in a documents substantiate this.

   To solve this problem, the paper loops over all nodes x in T_1, and calculates similarity with all *unmatched* nodes $y \in T_2$. The tuple with the highest similarity is picked to be added to matching M, if that similarity is high enough. So instead of just matching nodes when their similarity is high enough, they are only matched if their similarity is high enough *and* as high as possible for the current node.

The last change with regard to our explanation in Section 7.3, is that XMLDiff implements a variant of the Fast Matching algorithm. This alteration of the matching algorithm is explained in Section 5.3 of [11].

This implementation tries to fasten the matching algorithm by prepending a loop to it. This loop concerns all pairs of nodes (x, y) that occur in $lcs(T_1, T_2)$, and appends it to M'. Thereafter, the usual matching algorithm is performed. $lcs()$ computes a list of pairs that occur in the *longest common subsequence* of both trees. Two nodes are defined equal when their similarity is greater than or equal to 0.5. Of course, the computation of $equal_{inner}$ does not make any sense here, as none of the nodes are matched yet. So the similarity only depends on $equal_{leaf}$.

Taking into account these changes, the matching algorithm looks as follows:

```
1  M = []
2  mark all nodes in T1 and T2 as unmatched
3  for all pairs (x, y) in lcs(T1, T2)
4      M.append((x, y))
5      mark x, y as matched
6  for all unmatched nodes x in T1
7      best_match_value = 0
8      match_node = None
9      for all unmatched nodes y in T2
10         match_value = compare(x, y)
11         if match_value > best_match_value
12             best_match_value = match_value
13             match_node = y
14         if match_value = 1
15             break
16     if best_match_value > f
17         M.append((x, match_node))
18         mark x and match_node as matched
```

8.3 Generating the edit script

When we refer to line numbers in this Section, we refer to the algorithm shown in Section 7.4.3.

Because elements have four characteristics instead of two (Section 8.1), the library has three extra methods which are inserted into the algorithm

- $updateNodeAttributes(x, y)$. When the nodes of a match (x, y) are compared, the attributes of x must also be updated to have the values of the attributes of y. This function is created for that purpose. It performs a simple version of each step of the algorithm (Update, Align, Move, Insert, Delete) on all attributes. It starts by creating three lists: newKeys, removedKeys, and commonKeys. The first one holds attributes that do not occur in x, but do in y. The second one holds

attributes occurring in x but not in y. The third one holds attributes that occur in both nodes. Then for each step:

  - Update. for all keys k in commonKeys, it checks if the value u of that key in x is equal to the value v of that key in y. If $u \neq v$, it creates EditAction $UpdateAttribute(x, k, v)$: Update attribute k in node x to have value v.
  - Align. This step is skipped. The order of attributes does not matter in HTML.
  - Move. This step is used to update keys of attributes. For all keys k in removedKeys, v is the value of k in x. If v occurs as the value of a key $k' \neq k$ in y, create EditAction $RenameAttribute(x, k, k')$: Change the key of attribute k in x to k'.
  - Insert. For all key value pairs (k, v) in newKeys, create EditAction $InsertAttribute(x, k, v)$: Create attribute k with value v in x.
  - Delete. For all keys k in removedKeys, create EditAction $DeleteAttribute(x, k)$: Delete attribute k in x.

- $updateNodeTag(x, y)$. As we know from Section 8.2, the tags of x and y do not have to be equal for them to be matched. So when their tags differ, create EditAction $RenameNode(x, t)$: Update the tag of x to have value t, where t is the tag of y.

- $updateNodeText(x, y)$. This function creates the UPD(w, v(x)) operations in line 14. It creates $UpdateTextIn(x, i(y))$ and $UpdateTextAfter(x, t(y))$ to update x's inner text and tailing text if they differ from those of y. $i(y)$ is y's inner text, $t(y)$ is y's tailing text.

When a node is duplicated as in line 8, the library does not duplicate texts and attributes. Instead, an empty element is created having the same tag as the duplicated node. Later in the algorithm, text, tail, and attributes are added, each via its own EditAction. This way the user sees the creation of a node step by step.

Taking into account these changes, the edit script algorithm looks as follows:

```
E = []
M' = M
for all nodes x in bread_first_traverse(T2)
    y = p(x)//y in T2
    z = partner of y in M'//z in T1
    if x does not have a partner in M'
        k = find_position(x)
        e = InsertNode((w, l(x), ""), z, k)//w is a copy of x
        M'.append((w, x))
        mark w and x to be in_order
        E.append(e) and apply e to T1
    else if x is not the root
        w = partner of x//w in T1
        v = p(w)//v in T1
        if (v, y) not in M//The parents of w and x are not matched
            k = find_position(x)
            e = MoveNode(w, z, k)
            mark w and x to be in_order
            E.append(e) and apply e to T1
        if l(w) is not equal to l(x)
            e = RenameNode(w, l(x))
            E.append(e) and apply e to T1
    updateNodeAttributes(w, x)
    align_children(w, x)
    if i(w) is not equal to i(x)
        e = UpdateTextIn(w, i(x))
    if t(w) is not equal to t(x)
        e = UpdateTextAfter(w, t(x))
for all unmatched nodes x in post_order_traverse(T1)
    e = DeleteNode(x)
    E.append(e) and apply e to T1
```

Chapter 9

HDiff

As mentioned in Section 5.2, GX Software needs the end product to be in Java. We have therefore implemented the algorithm in Java, following the implementation of XMLDiff. This Section is dedicated to showing the differences between XMLDiff and HDiff, and to substantiate these choices.

The first addition is that we extend nodes with another attribute: the style attribute. In XMLDiff, this attribute is handled in the same way as other attributes: the value is a string. What we do now, is create a StyleAttribute for each key:value pair in the style attribute. The result is that we can compare styles using key value pairs instead of just strings.

We also split class attributes. Its values are not handled as strings, but rather as a list of classes. The algorithm can then count the number of equal classes when comparing nodes. This improves the similarity calculation when classes occur in different order. For example:
`<div class="div child element"/>` and
`<div class="child div element"/>` now have equal classes.

9.1 Finding matchings

This Section describes changes we have made to the matching algorithm.

1. We have implemented a function $bestMatch()$. As the name suggests, it does not only compute legitimate matches, but it computes the best possible matches. We prefer this function, as we have noticed that when using the original algorithm on documents found online, the $match()$ function produces "wrong" matches quite frequently. This happens because nodes in T_2 are no longer taken into account as soon as they are matched. To clarify, consider the simple example trees T_1 and T_2:

```
<html>
   <div id="el1" class="text-div child">
      First element with some text
   </div>
   <div id="el2" class="text-div child">
      Second element with some text
   </div>
</html>
```

```
<html>
   <div id="el3" class="text-div child">
      Second element with some text
   </div>
</html>
```

The matching algorithm considers el1 first. It checks for similarity with el3. They are not completely similar, but they have much in common. Their classes match, and their inner text is quite similar as well; they will thus be matched. When the function considers el2 in the next iteration, it will not check for similarity with el3 anymore, because el3 already has a partner. At the end, el1 and el3 will be matched, and el2 will not. However, it is obvious that a matching of el2 and el3 would result in a more natural edit script.

What we did to prevent these sub-optimal matches, is implement a solution for the Stable marriage problem [5]. As stated on wikipedia: "Given n men and m women, where each person has ranked all members of the opposite sex in order of preference, marry the men and women together such that there are no two people of opposite sex who would both rather have each other than their current partners. When there are no such pairs of people, the set of marriages is deemed stable.". Nodes in T_1 are men, nodes in T_2 are women. The preferences of men is defined by their similarity with women, and the other way around for women. "Married" nodes are nodes who occur in a match in M. When the solution to this problem is used instead of the original $match()$, the above "error" will not occur anymore!

So a new function is created called $matchIfCurrentBestMatch(x, y, value)$. Lines 6-8 in Section 7.3.2 are replaced with a call to this function. It is defined as follows:

```
1  u = x's current partner
2  v = y's current partner
3  if v is null:
4      if u is not null
5          unmatch x and u
6      match x and y
7  else if value > the similarity of v and y
8      unmatch v and y
9      if u not is null
10         unmatch x and u
11     match x and y
```

We then remove the word "unmatched" in line 4 in Section 7.3.2. So now all unmatched nodes in T_1 are compared to all nodes in T_2. The performance of this implementation is worse, but the resulting matching creates a more intuitive edit script.

2. We have updated $compare_{leaf}$. XMLDiff uses only inner text when performing this compare function. But in HTML, text is definitely not the only important factor when comparing nodes. It also has tailing text and attributes. On top of that, it now also has a map of StyleAttributes and a list of classes.

   When calculating similarity for of x and y, the algorithm now considers four values:

   - Text similarity and Tail similarity
     The similarity for inner text and tail is calculated in the same way.

     To calculate this value, the two strings of x and y are compared using their Levenshtein distance [3]. The similarity is given by $1 - \frac{d}{m}$. d is the distance of the two strings, and m is the maximum length of the two. This function is called $sim(s_1, s_2)$.
   - Style similarity and Attribute similarity
     For both of these maps, the similarity is computed equally.

     For all keys that occur in both x's and y's map, $sim(v_1, v_2)$ is computed for the values of v_1 of x and v_2 of y of this key. The results are summed, and divided by the total number of distinct keys.

     For the class attributes, an exception is made: its similarity is defined by the number of equal classes divided by the total number of distinct classes.

3. We have decided to keep the variable t in the algorithm. This gives the user a little more freedom in how to configure the program. The

main reason for this choice is that we can then use the enhancement described in the next point, which showed to improve the results.

4. As explained in Section 3.3.4 of [6], small changes in small sub trees can have a high impact. We do not want such a change to be the reason that its parent is not matched, as this can traverse upwards in the tree, resulting in large trees not being matched while they should. HDiff therefore applies the solution described in [6]: when an inner node has ≤ 4 descending leafs, the value of t is lowered to 0.4 for that tree. This is obviously in contradiction with the paper of Chawathe et al, but it works better on HTML.

5. Some HTML elements require a special case when comparing them with other nodes. A good example is the `<img>` element. In many cases, its only identifiable attribute is the src attribute. For two different images on the same website, the values of these attributes may be very alike, although the image is entirely different.

   For these type of cases, another function is added: $couldBeMatched(x, y)$. This function indicates whether it is possible that x and y could be matched at all. Its return value, a Boolean, is based on simple checks. Regarding the example above: When both nodes have the `<img>` tag, they can only be matched if the values of the src attribute are identical.

   The result of this function is checked when calculating the similarity of two leafs. If it returns false, a similarity of 0 is returned. If the result is true, the function continues like it used to. This makes the matching algorithm not only more accurate, but also more efficient.

6. To increase the precision of matches, simple heuristics can be added. Examples of these heuristics are: increase the similarity of two nodes by 20% if their tags are equal; if elements are hard to identify, e.g. they have few characteristics, increase their similarity by 25% if their parents are matched.

   These multipliers decrease the chances of simple leafs being matched to other simple leafs where it is clear that they should not be.

   After the similarity is calculated, it is multiplied by the result of $getSimilarityMultiplier(x, y)$. This function implements simple heuristics as the examples above.

7. For some documents, texts and tails are more important in similarity calculation than attributes and styles. An example is a static HTML file that mainly consists simple `<p>`'s and `<div>`'s containing large parts of text. Classes and other attributes will probably occur, but will not have much to do with the layout of the file. In that case, the

matching is more accurate if the similarity is based more on text and tail than on attributes and styles.

Therefore, the user is now given the option to work with weights. There exist three weights: TextWeight, AttributeWeight, and StyleWeight. When the four similarities of above are calculated, their values are multiplied by the corresponding weight (Text and Tail similarity both use the TextWeight). In the end, all values are summed and divided by the sum of *the used* weights. When one increases the value of TextWeight, the similarity of two nodes is based more heavily on their text and tail similarities, and less on attributes and styles.

Note that a weight is only used if the characteristic occurs in at least one of the nodes. If for example both nodes do not have any HTML attributes, the AttributeWeight is not used. The result is then given by the sum of the style and text divided by the sum of the TextWeight and StyleWeight. If none of the characteristics occur, -1 is returned as similarity. This notifies the matching algorithm that the similarity of the nodes is in this case only given by the $compare_{inner}$, and thus $compare_{leaf}$ is ignored. This is desired behaviour, as we can not say anything meaningful about the similarity of two nodes that do not have any identifiers (except for the similarity of their parents, which case is already handled by the heuristics of point 5.).

If we would not discard unused weights, the similarity of nodes with few to none characteristics would be very low to zero in any case. Take the common example of a `<table>` element without any attributes, styles, or texts, but with a large number of children. When calculating $compare_{leaf}$ (recall from Section 8.2 that this function is also computed on inner nodes) using all weights, the result would be 0, as we have no similarities between all characteristic. As $compare_{leaf} \leq f$, the table elements would never match. It is clear that if all children of the table match, the desired behaviour is that the tables would also be matched.

8. In many documents, *unidentifiable leafs* exist. Examples are `<br/>` and `<span></span>`. These nodes are easily mismatched to unidentifiable leafs in other parents in the tree, despite the improvements described above.

   Because these leafs say little to nothing about the document, we exclude them from the algorithm. If an element is such a leaf, it is skipped in both the matching and the edit script part. The output therefore contains all the unidentifiable leafs of T_1, and none of the ones of T_2.

9.2 Generating the edit script

This Section describes changes regarding the generation of the edit script.

1. In Section 8.3 we talked about the function $updateNodeAttributes(x, y)$. This function is now performed by a dedicated class called AttributeDiffer. This AttributeDiffer creates a special case for the value of the "style" attribute. It skips this attribute itself, while another class, the StyleDiffer, is responsible for comparing this attribute. It does essentially the same thing as the AttributeDiffer, but on the value of "style".

   We have split these two cases, such that the algorithm can create separate EditActions for them. Therefore the actions InsertStyleAttribute, DeleteStyleAttribute, RenameStyleAttribute, and UpdateStyleAttribute are added. They have similar meanings as the ones explained in Section 8.3.

   With the separation of these actions, the algorithm can now show the user the difference between the change of a normal attribute, like a class or custom attribute, and the change of a certain style element, like color or size.

2. Recall line 8 of the algorithm described in section 8.3. This is the point where a copy is made of a node in T_2. XMLDiff only copied the tag of node x at this point. Later in the algorithm, other characteristics like text and attributes were added.

   We have made nodes cloneable. When element x is cloned to element w, all characteristics of w are equal to the ones of x, except for the parent. So it creates w by cloning x, and afterwards set z to be the parent of w. It also recursively matches x and w along with their children, and marks them as in order.

   The result is that it creates only one EditAction for a newly inserted node. We prefer this over the solution of XMLDiff, because it makes the result less messy. Instead of creating an EditAction for each single characteristic, plus one for the insertion of the node itself, it only creates one EditAction for the insertion itself. It should be clear to the user that all characteristics were inserted along with that node.

3. We have also extended the way that changes of inner texts and tails are shown to the user. Recall the explanation of HtmlDiff in Section 5.2. The algorithm consists of three parts: Parsing the input, calculating the differences, and creating the output. In our case, we can obviously skip the part where the input is parsed. But we also noticed that the main problem of this library is the way the input is parsed. So we decided to implement the last two parts of the library, where the elements are characters in texts or tails.

XMLDiff just marks a text or tail to be "changed" when two values are not exactly equal. In large similar texts, it is worth the effort to show these differences in more subtle way. Thus we have implemented the SequenceMatcher used in HtmlDiff, and apply the opcodes to inner texts and tails. This results in `<ins>` and `<del>` tags inside texts. This visualizes the precise differences to the user, instead of showing a block of text marked as "changed". We have now got a solution combining Sections 3.1.1 and 3.1.3.

The SequenceMatcher is only used when the similarity between the two strings is greater than or equal to 75%. If it is smaller, it marks the whole first text as deleted and the whole second text as inserted. If this check would not be added, very dissimilar texts would result in a very messy result with a lot of `<ins>`'s and `<del>`'s.

9.3 Other changes

This Section describes changes we have made regarding the structure of certain elements of the algorithm. They are mostly made such that the output that is created is more applicable to this project.

1. When an element is deleted, all EditActions that belong to the children of this element are also deleted. The only possible EditActions of these children are DeleteNode actions, and when the user sees that a parent is deleted, it should be clear that all of its children are deleted as well. So to reduce messiness, deletions of all of its children are removed, and thus not shown.

   To show that we are certain that all actions of the children are DeleteNode actions, assume the following:

   (a) The parent being currently deleted is node $x \in T_1$.

   (b) x has child $u \in T_1$, and thus x is u's parent.

   (c) Assume u has EditAction e.

   (d) From 1a we derive that x does not have a partner, because if it would, it would not be deleted.

   (e) If e were an InsertNode action, u should have a partner $v \in T_1$, whose parent $y \in T_2$ would be the partner of u's parent (by definition of the Insert part of the algorithm). But we know from 1b and 1d. that u's parent does not have a partner. So we know that e is not an InsertNode action.

   Now if e would be any other action than the DeleteNode action (this is the assumption):

i. We can assume that u has a partner $v \in T_2$, because any actions on u not being DeleteNode or InsertNode actions, are changes with respect another node: its match v. And we know from 1e and the assumption that e is not and DeleteNode or InsertNode Action.

ii. From 1d and 1i we derive that If v has a parent y, this parent is not a partner of x.

iii. From lines 1b and 1ii we derive that $p(v) \neq p(u)$.

iv. From the fact that the algorithm proceeds bottom up, we know that all EditActions on u are created and applied before any actions on its parent x.

v. From lines 1iii and 1iv we derive that if v has parent y, then $y \neq x$ and u is already moved to that parent. And if v does not have a parent, u is already set to be the root, as v is the root too.

vi. From 1v we derive that either u is the root and thus does not have a parent, or u has a parent which is not x.

vii. 1vi contradicts 1b, thus the assumption cannot be true!

2. We have added a new class called TreeEditor. Whenever an EditAction is created, it is applied by this editor. Before applying it, it creates a so called "PopupText" and adds it to the corresponding node. These texts describe the changes of the actions. Examples are "This node is deleted" and "The style attribute with key k has its key changed to key k'".

3. The class EditAction has a function called $isVisible()$. This function defines whether the PopupText belonging to the action should be shown in the output file. We have added this function because for some actions we are certain that they have no visual effect. Examples are actions with a target that has a tag that is never visible, like `<script>` and `<base>`. But also elements with the style attribute visibility:gone are never to be shown to the user.

   It is hard if not impossible to handle all cases in this function. For some EditActions we are not sure whether it has a visible effect or not. An example is the addition of a class, which could or could not result in new css attributes, changing the layout of the node.

4. We have added a class Outputter. This class is responsible for creating an output HTML file that shows the user all calculated differences between T_1 and T_2. It creates the output from a template, which includes a `styles.css` file and a `script.js` file. It also adds an element `<div id="explanation"/>`. The JavaScript ensures that whenever

the user hovers a changed element, the PopupTexts of this element are shown as a list inside the explanation div.

The Outputter then performs the function `write(n)` on the root:

```
Add 'edited' and 'deleted' classes to n if needed
Write the tag of n, its attributes, style attributes,
  and 'popuptext' attribute
Write the inner text of n
for all children c of n
    write(c)
Write the closing tag of n
Write the tail of n
```

The attribute "popuptext" is a `<ul>` holding a `<li>` for each PopupText of the node. It finishes up by properly closing and saving the html file.

5. When a node is moved, a clone is created. The clone is being marked "deleted", and gets the popuptext "This node is moved". It is placed in its parent at the old position. The original node is moved to the new position. This way, a moved node is shown in both its old and its new position. When nodes are moved to a whole different location in the tree, this helps the user in tracking down the old position of a moved node.

6. As mentioned in the abstract, GX needs to apply our software on all pages of a complete website. We have therefore created a Snapshotter. To run, it needs four values:

   - URL u_l of the login page.
   - Login credentials c.
   - URL u_d of the page which shows, after the user is logged in, a list of all links in the complete website
   - Key k that is used to uniquely identify each URL. When no key is provided, the inner text of a node is used as its unique identifier.
   - Folder name f. The snapshot is saved in this folder.

   The Snapshotter then makes a snapshot of the complete website. It starts by visiting u_l, and posting the login form with the provided credentials c. After this login, it has the right cookies such that it is allowed to visit u_d.

   It then visits u_d, and crawls all `<a>` elements of the page. It saves these in a map. For each found element, it maps the value of the identifying key k to the value of the href attribute.

It loops over all entries in the map. Using a WebClient, it visits the URL of each entry. We use a WebClient such that we get the well-formed resulting DOM, after the JavaScript is executed. As mentioned in Chapter 2, this is exactly the HTML we need to compare. For each resulting HTML, a file is saved in folder f with the name equal to the key of the entry.

After all entries are visited, we have saved a complete snapshot of the website, were each page has an associated HTML file in folder f.

9.4 Example run

Now that we have got a working implementation, we run it with the same input as we did with DaisyDiff and HtmlDiff. Figure 9.1 shows the result. Note the following:

- Because the input files were retrieved via web.archive.org, all images had different src attributes on both pages, even if it was the same image. This happens because the base of each src referred to the snapshot saved by web.archive.org. This also explains why DaisyDiff had a hard time matching images.

  To prevent our algorithm from not matching any images, we removed the requirement that images can only be matched if their sources are identical (item 5 of Section 9.1). This example did not result in many mismatches due to the absence of this requirement. So now the images are matched, and their difference is shown as a change in the value of the src attribute.

- We have excluded changes in "href" attributes in this example. The reason is the same as for "src": each href is marked as changed, because of the base being in web.archive.org.

- Our variables f and t were set to 0.7 and 0.6 respectively.

Figure 9.1: Example output of HDiff

Chapter 10

Future work & conclusion

As seen in Chapter 9, we have made quite a few changes and enhancements on the original paper an XMLDiff. Nevertheless, there are still improvements possible. These are split in two types. The first type of improvements are the ones than improve the library in a way that it generates better output. Improvements of the second type are focused on the implementation of GX. When GX is going to implement the software in their systems, it is advised that they implement these improvements first.

10.1 Improvements to the algorithm

1. In item 5 of Section 9.1 we talked about the function *couldBeMatched*(). This function currently only checks whether the src attributes of `<img>` elements are identical. When the software is used in practice, one will find other checks that can be added. The output will improve because false matches will occur less often, and the efficiency will benefit from it as the $compare_{leaf}$ will have to be ran less often.

2. Item 6 of Section 9.1 could also use more cases. The best way to find out which cases result in a better matching, is to test the library on real world examples. Complicated HTML inputs might result in weird matches, which can be avoided by adding multipliers in this function. This clearly applies mostly to elements with few characteristics, as these are most vulnerable to be matched to the wrong partner.

3. To reduce messiness in the output, the function *isVisible*() of item 3 of Section 9.3 should be enhanced as well.

   It currently only checks the tag of the target. If this tag is a tag that is invisible by definition, false is returned. More sophisticated checks can and should be added.

An example we have seen is that some pages have a time stamp relative to the current time saved in an attribute in the `<body>` element. It is clear that this value changes for each snapshot, and that this change does not have a visual effect. On top of that, because it is a change in the `<body>` element and thus occurs at one of the highest levels in the tree, the whole screen will marked as "changed".

It is hard to verify that these type of changes are invisible, but with enough test cases one might find general and logical checks to add to this function.

4. Depending on the case on which the library is used, one might want to add support for CSS changes. Included CSS files are currently not handled by the library, while some cases might need this.

   For CSS support, one could proceed in two ways.

   - It is possible to rewrite all entries in the CSS file to style elements for the according HTML elements. This is the easiest way to add CSS support, as tools yet exist for this purpose. Because the library already has separate cases for the style element, it will then work directly. However, it might slow down the algorithm drastically, as elements get very long style values.
   - It might therefore be better to write a separate Differ for imported CSS style sheets. The changes to these sheets are then separated from changes directly to HTML, which offers the possibility to show these changes to the user in a different way.

10.2 Improvements for GX Software

1. The first improvement is efficiency. When one would want to run the Snapshotter twice on a whole website, and then run the Differ on all resulting files, this might take too long. In our test case, it took about a second per page to snapshot. When a website has several hundreds of thousands of pages, it is not doable to snapshot all of these. Depending on the situation, it might be sufficient to take a sample of about a thousand pages, and check these for differences. This is of course not a improvement of efficiency of the algorithm, but it will reduce the time needed to check a website for differences.

   One might also want to drop the $bestMatch()$ function as explained in item 1 of Section 9.1. This function definitely improves the result, but this might not be needed in practice. If the regular $match()$ function generates sufficient results, this is a better choice as it is more efficient.

2. When the software is used to compare a whole website, the result needs to be shown to the user in a user friendly way. This typically consists of looping over all files that have changed, highlighting each change separately. The user then has to confirm each change. When all changes are confirmed by the user, the software confirms that the website looks as its supposed to.

   Firstly, this shows us the importance of item 3 of Section 10.1: when a user has to confirm all sorts of changes that have no visual effect, this reduces the value of the software.

   But more importantly, this raises another problem. Our test case ran on a website where the title of each page occurred in (a sub menu of) the menu, which was positioned at the left of the screen. In our second snapshot, the title of one of the pages was changed. This resulted in the snapshot of every page being changed. Why? The new title resulted in a change in the menu bar, which naturally occurred on every page of the website.

   So when the software is used in practice, we need some observer that recognizes *equal or similar* changes that occur on a significant part of all pages of the website. The user should then confirm the change of that part only on the first page. The software then knows that the change of the menu on all other pages is already confirmed, such that it does not have to bother the user with that change anymore.

   A menu is not the only element on which this issue occurs. We also have headers and footers, and included elements of other websites that occur on every page. This observer should be quite sophisticated, as we do not want it to skip changes that the user has not confirmed yet.

3. Since GX uses this software in their Content Management System XperienCentral[1], they have control over the format of the HTML. This is a great advantage, which they could use to improve the algorithm.

   For example, we could handle our problem of item 2 in another way. GX knows which elements on which pages have which function: they know where the footer and menu are. They can then easily tell the algorithm to skip elements in the menu, such that these changes do not occur on every page.

   They could also enforce each element to have automatically generated unique identifiers. In Section 3.1.3 we talked about this option. We ruled out the possibility that we would use identifiers to match nodes, as the documents on which the algorithm is used on most likely will not have them. If GX can ensure that all elements on

[1] `https://www.gxsoftware.com/en/products/xperiencentral.htm`

the whole website have unique identifiers, they should definitely use these. The matching algorithm will become way more efficient, and will be able to calculate fully accurate matches.

10.3 Conclusion

The current version of our software is definitely able to correctly compute difference in two HTML documents.

However, we have seen that one of the hardest parts is to define whether a change has a visual effect to the user.

It also became clear that just computing differences between two documents is not sufficient when comparing a whole website. We need these changes to have a place in the comparison of the complete website. A Differ between Differences might be desired. When a few changes are very alike, they can be grouped into a group of changes, which can then be confirmed by the user at once.

So yes, we can effectively compute and show *differences* between two HTML objects. What still has to be sorted out is if we can correctly distinguish *visible* from *invisible* changes, and if we could place changes of single pages into the context of the complete website.

Bibliography

[1] Difflib. https://docs.python.org/2/library/difflib.html.

[2] XMLDiff. https://xmldiff.readthedocs.io/.

[3] Levenshtein distance. https://en.wikipedia.org/wiki/Levenshtein_distance, 2018.

[4] Longest common subsequence problem. https://en.wikipedia.org/wiki/Longest_common_subsequence_problem, 2018.

[5] Stable marriage problem. https://en.wikipedia.org/wiki/Stable_marriage_problem, 2018.

[6] Martin Pinzger Harald C. Gall Beat Fluri, Michael Würsch. Change distilling: Tree differencing for fine-grained source code change extraction. 2007. www.merlin.uzh.ch/contributionDocument/download/2162.

[7] Richard Cyganiak. Htmldiff. https://github.com/cygri/htmldiff, 2011 – 2017.

[8] Google. Daisydiff. https://code.google.com/archive/p/daisydiff, 2007.

[9] David E. Metzener John W. Ratcliff. Pattern matching: the gestalt approach. http://collaboration.cmc.ec.gc.ca/science/rpn/biblio/ddj/Website/articles/DDJ/1988/8807/8807c/8807c.htm.

[10] Rick Walker Ilir Jusufi Charles D. Hansen Jonathan C. Roberts Michael Gleicher, Danielle Albers. Visual comparison for information visualization. 2011. https://pdfs.semanticscholar.org/a5e3/cf9f7bfdbc227673a6d8f4d59112f1b5bb3a.pdf.

[11] Hector Garcia-Molina Jennifer Widom Sudarshan S. Chawathe, Anand Rajaraman. Change detection in hierarchically structured information. 1996. http://ilpubs.stanford.edu:8090/115/1/1995-46.pdf.

Appendix

Listing 1: example_T_1.html

```
<!DOCTYPE html>
<html lang="en">
    <body>
        <div>
            <p>
                This is the first paragraph of the example file.
            </p>
            <p>
                This is the second paragraph of the example file.
            </p>
        </div>
        <ul>
            <li>
                This is the first item of the list.
            </li>
            <li>
                This is the second item of the list.
            </li>
            <li>
                This is the third item of the list.
            </li>
        </ul>
        <figure>
            <img style="display:block"
                 src="https://www.google.com/images/branding/
                     googlelogo/2x/googlelogo_color_272x92dp.png"/
                    >
            <span>
                This is a text below an image of google
            </span>
        </figure>
        <a id="link" href="https://www.google.com">
            <span>
                This is a link to google
            </span>
        </a>
    </body>
</html>
```

Listing 2: example_T_2.html

```
<!DOCTYPE html>
<html lang="en">
    <body>
        <div>
            <p>
                This is the first paragraph of the example file.
            </p>
            <span>
                This is the first item of the list.
            </span>
            <p>
                This is the second paragraph of the example file.
            </p>
        </div>
        <ul>
            <li>
                This is the second item of the list.
            </li>
        </ul>
        <figure>
            <span>
                This is a text below an image of google
            </span>
            <img style="display:block"
                 src="https://www.google.com/images/branding/
                     googlelogo/2x/googlelogo_color_272x92dp.png"/
                     >
        </figure>
        <a id="link" href="https://www.ru.nl">
            <span>This is a link to the website of Radboud</span>
            <span> &#x1f517;</span>
        </a>
    </body>
</html>
```

Figure 1: example_T_1.html and example_T_2.html visualized side-by-side