Extracting variant data from templatized web pages

Research and Development Project Report

Naman Agarwal

Nishant Totla

Guided by:
Prof. S. Sudarshan

Department of Computer Science and Engineering
Indian Institute of Technology
Mumbai

April 28, 2011

1 Introduction

Most web pages on different web sites today consist of text/images which convey relelvant information interspersed with text/images that appear as part of the structure of a particular website. By text/images appearing as part of structure we refer to parts of the website that serve as links to other pages or for example text which is a motto of the website. We expect that when a user enters a particular search query, he is often more interested in the information containing parts of a website as compared to this structural data. Such structural data often appears as part of a static template governing the set of most pages in a website and can be deciphered by analyzing the areas of similarity in a set of representative pages. This is what we aim to do in our project. We first build a Pattern Tree (a structure for holding the template governing the web site) from a representative set of pages from a web site and further use the Pattern Tree to extract informative data from all the web pages of a website. We further index webpages based only on the informative data hence extracted and run simple search queries in this system. The following sections proceed by defining the problem followed by descriptions of algorithms for tree merging and alignment that we put to use and further an empirical analysis of the system's performance on a few sites.

2 Problem Definition

The first step towards the template detection problem is to define an appropriate structure for holding the template. The structure we assume is very similar to the one used in [2]. The structure used is called **PatternTree**. The structure PatternTree is an abstract representative of the HTML trees and subtrees and hence is structured in a tree like format. The structure has 3 primary members -

1. **Symbol** - holds the tag node of the root of the corresponding HTML Tree.
2. **Type** - holds a particular property of the node which defines its repetitions/optionality . It can take the following values -
   (a) fixed - this suggests that the node necessarily appears in the template and there is just exactly one occurence of this node.
   (b) set - this suggests that at least one occurence of this node is necessarily present, however it can be followed by as many occurences of such nodes.
   (c) optional - this suggests that the occurence of this node is optional i.e. may occur once or not at all.
   (d) variant - a node of variant type contains data that we want to pick up from the page.
3. **childList** - holds a list of children of the node. The children are themselves PatternTrees, hence generating the tree-like structure of a PatternTree.

As an example, a possible PatternTree for the template generating two HTML trees is shown in Figure 1. We now define the language of a patternTree P.

$$Language\ of\ P = Language\ of\ the\ regular\ ExpressionReg(P)$$

$$Reg(P) = \begin{cases} P.text & \text{if P.type = variant} \\ < P.symbol > ChildLang(P) < P.symbol > & \text{if P.type = fixed} \\ (ChildLang(P))? & \text{if P.type = optional} \\ (ChildLang(P))+ & \text{if P.type = set} \end{cases}$$

$$\textbf{where } ChildLang(P) = Reg(P_{C1}).Reg(P_{C2})......Reg(P_{Cn})$$
$$P_{C1}, P_{C2}, ..., P_{Cn}\ are\ children\ of\ the\ patternTree\ P$$

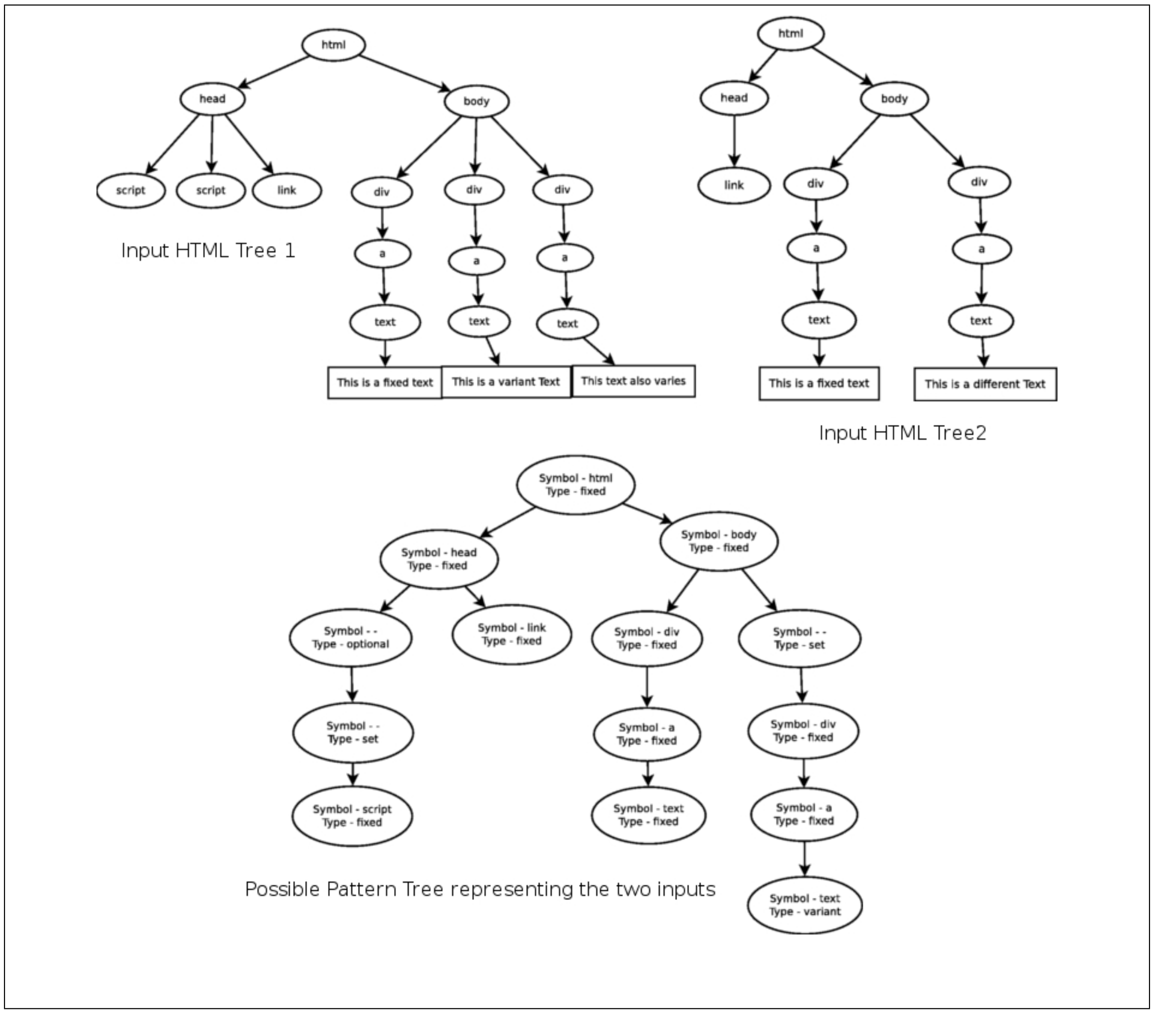


Figure 1: Example of DOMTrees and corresponding PatternTrees

Given the above definition of a language of a Pattern Tree, the problem that we wish to tackle is to come up with a good estimate of a patternTree P such that all given seed HTML trees belong to the language of P. Also once the patternTree is created, we hope to match it with more input HTML trees being generated from the same template and be able to pick out the 'variant' data i.e. data that changes across pages.

3 Approach

We have broadly followed the approach described in [2]. While the algorithm is similar, we have made changes which we thought would help us get better patterns. The problem is two fold.

1. Find pattern tree
2. Extract data from a given DOM Tree using this pattern tree

We shall describe the complete algorithm, with examples to demonstrate the working of every part. The pattern tree is found by a cover algorithm called the Tree Merging Algorithm (Section 3) and data from a DOM tree is retrieved using the algorithm described in Section 8.

4 Tree Merging Algorithm

This is the main part of the algorithm that generates a pattern tree (like the Pattern Tree structure described in Figure 1). Given a set of DOM Trees T, all having a common root symbol (e.g. html at the top level), Procedure 1 will return a Pattern Tree structure that entails all trees in T. We then expect this Pattern Tree structure to extrapolate to DOM Trees of more pages (other than those trained on) from the website, so that we can extract variant data. We now briefly describe how the algorithm proceeds.

There are four major steps:

1. Peer node recognition
2. Matrix alignment
3. Matrix compression
4. Pattern mining

A peer matrix M is created as described in Section 4. A Symbol Table S is also created. S is needed because trees are represented by symbols, and the reverse mapping needs to be done later. After this, a *childList* is obtained by doing Matrix Alignment (Section 5) and Pattern Mining (Section 7). *childList* ultimately contains symbols, and represents what is the expected most general pattern at that level. Once childList is obtained, we then find the Pattern tree that represents each symbol (by a recursive call to the the Tree Merging algorithm), and then combine the first level Pattern trees to get the main Pattern tree that entails DOM Trees of T.

Procedure 1 Multiple Tree Merge

```
Input: T ← List of DOM Trees, P ← Common Root Symbol
Output: PatternTree ← Pattern tree entailing all trees of T
  S ⇐ new SymbolTable
  M ⇐ recognizePeerNodes(T,S)
  childList ⇐ matrixAlignment(M,S)
  reference ⇐ neoPatternMining(childList)
  childList ⇐ preOrderForList(childList)
  patternChildList ⇐ new List of Pattern trees
  for all $i \Rightarrow 0 : size(childList)$ do
    sym ⇐ childList(i)
    if sym is a variant node then
      if sym is a data node then
        p ⇐ new Pattern tree (FixedData)
        patternChildList.add(p)
      else
        toAdd ⇐ list of trees corresponding to sym
        p1 ⇐ multipleTreeMerge(toAdd,rootSymbol(sym))
        patternChildList.add(p)
      end if
    else
      p ⇐ new Pattern tree (VariantData)
      patternChildList.add(p)
    end if
  end for
  patternHead ⇐ new Pattern tree
  return buildPatternTree(reference,patternHead,patternChildList)
```

5 Peer Node Recognition

Peer Node Recognition is the problem of recognizing whether two nodes at a level are alike or not. This is the first step towards template recognition, as during peer node recognition the algorithm marks two children at the same level in two different trees as the same, laying down the basis for further alignment and pattern extraction. Our treatment of peer node recognition is in essence the same as the treatment in [2], i.e. to decide whether two nodes are alike are not we use the tree alignment algorithm described in [3] and we normalize the score the same way as [2].

However unlike [2] which brands nodes as alike or not alike, we try and distinguish nodes also on the basis of whether they are the same structurally or textually. Structural similarity refers to similarity dependent only on the html structure that resides in the subtree rooted at the node in contention. More simply put, structural similarity ignores the text while doing the matching. Textual similarity on the other hand refers to matching nodes based on their text also, i.e. it does take into account how well the text nodes at the leaf level match also.

We perform the matching using the Procedure 2. The argument flag indicates whether or not leaf nodes are to be taken into consideration. Further during peer node recognition we assign symbols to all the nodes to act as identifiers and also to encode the similarity information gathered thus far.The algorithm for the recognition and assignment of symbols is described here. Symbols are encoded as $\mathbf{S}_k\mathbf{T}_l$ where k and l are integers. Two nodes are structurally same if their k values match and they match textually when both their k and l values match.

As an intuition to why such a differentiation between structural and textual similarity was made, for now, we can vaguely state that two nodes that are structurally and textually similar signal a constant structure in the template, where as nodes which structurally similar but textually different are indicators of regions where the variant data resides. More involved treatment, with examples is done in Section 6.

6 Matrix Alignment

The Peer Node recognition step sets up the Peer Matrix M. M contains symbols $\mathbf{S}_k\mathbf{T}_l$ where k and l are integers. It has the following structure:

1. There are as many columns as the number of DOM Trees whose common pattern we are trying to find.
2. The i^{th} symbol in the j^{th} column is the $\mathbf{S}_k\mathbf{T}_l$ symbol that the Peer Node Recognition step assigned to the i^{th} first level subtree (child) of the j^{th} DOM Tree.

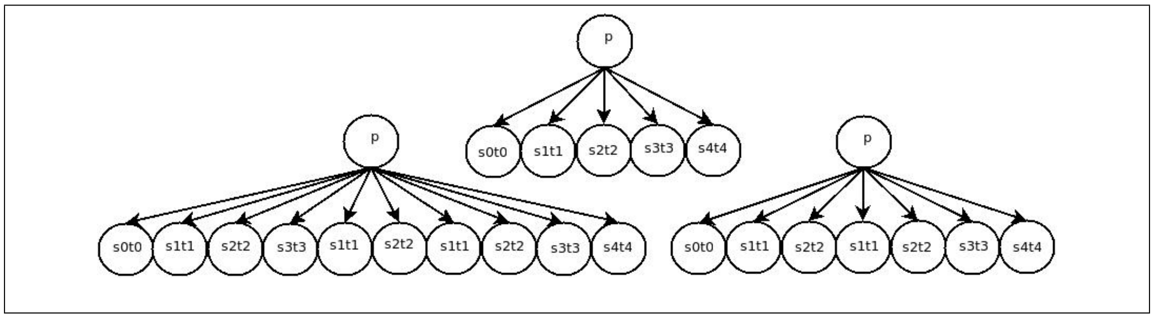


Figure 2: Peer nodes for three trees (Source: [2])

In Figure 2, we have presented an example of the output of the Peer Node Recognition step for an example with three trees. The first level subtrees have been assigned symbols based on their structural and textual content. The resulting Peer Matrix is shown in Figure 3 (a).

Note that for each symbol in M, the first level subtrees of each DOM Tree have been maintained separately. This is required later when the main algorithm descends into lower levels. But at the alignment stage, we do not look at the lower structure.

Procedure 2 Given two DOM Trees return how well they match structurally/textually

```
Input: $A, B \leftarrow DOM\ Tree, flag \leftarrow boolean$
Output: $score \leftarrow int$
  if $A.symbol \neq B.symbol$ then
    return 0
  else if $flag$ & $isText(A.symbol)$ then
    if $A.symbol \neq B.symbol$ then
      return 0
    end if
  else
    $m \Leftarrow A.arity$
    $n \Leftarrow B.arity$
    $M \Leftarrow array(m+1, n+1)int$
  end if
  for $k = 0 \rightarrow m+1$ do
    $M[k][0] \Rightarrow 0$
  end for
  for $k = 0 \rightarrow m+1$ do
    $M[0][k] \Rightarrow 0$
  end for
  for $i = 0 \rightarrow m+1$ do
    for $j = 0 \rightarrow m+1$ do
      $x \Rightarrow simpleTreeMatching(A.children[i-1], B.children[j-1], flag)$
      $t \Rightarrow max(M[i-1][j], M[i][j-1])$
      $t \Rightarrow max(t, M[i-1][j-1] + x)$
      $M[i][j] \Rightarrow t$
    end for
  end for
  return M[m][n]+1
```

During alignment, we shift columns till we reach an aligned matrix. This shifting might introduce certain empty blank spaces too in the matrix. We say that M is aligned when each row is aligned. A row is aligned when all its non-blank symbols are the same. A row may also be aligned when all its non-blank symbols are data (text/image) nodes, even if they are different symbols. This distinction is essential in picking out variant nodes, as we shall see later.

As shown in Procedure 3, the alignment algorithm tries to align each row successively, till the entire Matrix gets aligned.

Procedure 3 Matrix Alignment

```
Input: $M \leftarrow PeerMatrix, S \leftarrow SymbolTable$
Output: $childList \leftarrow ListOfChildren$
  Spans ⇐ computeSpans(M,S)
  bNodes ⇐ null
  row ⇐ 0
  while !$Aligned(M)$ do
    while !$AlignedRow(row, M)$ do
      shiftColumn ⇐ getShiftColumn(row,M,spans,S,shiftLength)
      if $shiftColumn == -2$ OR $shiftColumn == -3$ then
        break
      end if
      makeShift(row, shiftColumn, shiftLength, M)
    end while
    row ⇐ row+1
  end while
  compressMatrix(M,S)
  bNodes ⇐ listAllVariantNodes(M,S)
  childList ⇐ getAlignmentResult(M,bNodes,S)
  return childList
```

The algorithm uses the Peer Matrix M, and also the Symbol Table S. It returns a childList, which is essentially a list of the aligned symbol for every row. This in a way represents the common structure at depth 1, that the algorithm has been able to mark out, for further processing.

The function *alignedRow* checks if the current row is aligned. If not, it tries to align the row, otherwise moves on to the next one, until M is completely aligned. The function *getShiftColumn* marks out the column that should be shifted down in the current row. Then all nodes in that column are shifted down by a distance *shiftLength*. The functions *alignedRow* and *getShiftColumn* need to be explained in detail.

The function *alignedRow* returns **true** in two cases. Firstly, if all nodes in a row of M have the same symbol (barring blank nodes). Secondly, if the nodes in the row do not have the same symbol, but are basic data (text/image) type nodes. This part, as can be seen, is one of the ways in which variant data nodes are picked up at the current level of processing. These rows are later marked as variant, and returned for further processing.

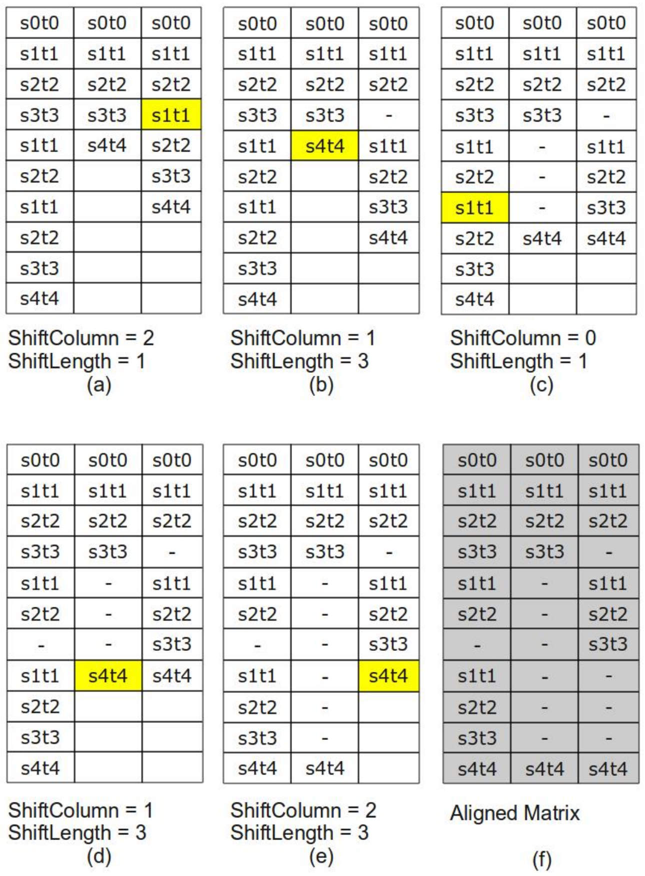
The function *getShiftColumn* is crucial to the alignment step. Before that, it must be noted that we have a notion of the *span* of a symbol, which is the same as that used by [2]. The function *getShiftColumn* takes a row r which is not yet aligned, and applies the following rules in order:

- (R1) Select from left to right, a column c such that the expected appearance of the node n ($= M[r][c]$) is not reached, i.e. there exists a node with the same symbol at some upper row r_{up} ($r_{up} < r$), where $M[r_{up}][c'] = n$ for some c', and $r - r_{up} < span(n)$. Then *shiftColumn* will be equal to c and *shiftLength* will be 1.

- (R2) If no column satisfies the condition of R1, then we select a column c with the nearest row r_{down} ($r_{down} > r$) from r such that $M[r_{down}][c'] = M[r][c]$ for some $c' \neq c$. In this case, *shiftLength* will be $r_{down} - r$.

- (R3) If both rules R1 and R2 fail, we then check if the r contains all data (text/image) nodes. If it does, then we specially set $shiftColumn$ to -2, and no shifting gets done. This row is considered aligned.
- (R4) If all of R1, R2 and R3 fail, then we select the symbol that occurs the maximum number of times on this row (if there are more than one, select any one). Keeping all columns with that symbol in r unchanged, we shift all other columns down by one. $shiftColumn$ is specially set to -3 after handling this case, and no more shifting is needed to align r.

Note that we have modified the rules of *getShiftColumn* used in [2]. While R1 and R2 are exactly the same, R3 and R4 are new. The advantage of using R3 in its current form is that some variant data nodes can be detected right at this step. When a row has all different data nodes, it seems logical to consider the row aligned, and not shift anything (of course, crucial to doing this is rules R1 and R2 failing, else such a data node might match something else in some other row). As for R4, it is the last resort. When all of R1, R2 and R3 fail, there is nothing that can be done with this row, and we must assume that structures on this row can occur in any tree. But we can't say anything for sure. We use the heuristic of keeping the most frequently occuring symbol on the row, in that row. This might give inaccurate patterns later. But as described in Section 6, all these rows will get marked as **optional**. This prevents any problems in data extraction (Section 8), which is the key goal.



s0t0	s0t0	s0t0
s1t1	s1t1	s1t1
s2t2	s2t2	s2t2
s3t3	s3t3	s1t1
s1t1	s4t4	s2t2
s2t2		s3t3
s1t1		s4t4
s2t2		
s3t3		
s4t4		

ShiftColumn = 2
ShiftLength = 1
(a)

s0t0	s0t0	s0t0
s1t1	s1t1	s1t1
s2t2	s2t2	s2t2
s3t3	s3t3	-
s1t1	s4t4	s1t1
s2t2		s2t2
s1t1		s3t3
s2t2		s4t4
s3t3		
s4t4		

ShiftColumn = 1
ShiftLength = 3
(b)

s0t0	s0t0	s0t0
s1t1	s1t1	s1t1
s2t2	s2t2	s2t2
s3t3	s3t3	-
s1t1	-	s1t1
s2t2	-	s2t2
s1t1	-	s3t3
s2t2	s4t4	s4t4
s3t3		
s4t4		

ShiftColumn = 0
ShiftLength = 1
(c)

s0t0	s0t0	s0t0
s1t1	s1t1	s1t1
s2t2	s2t2	s2t2
s3t3	s3t3	-
s1t1	-	s1t1
s2t2	-	s2t2
-	-	s3t3
s1t1	s4t4	s4t4
s2t2		
s3t3		
s4t4		

ShiftColumn = 1
ShiftLength = 3
(d)

s0t0	s0t0	s0t0
s1t1	s1t1	s1t1
s2t2	s2t2	s2t2
s3t3	s3t3	-
s1t1	-	s1t1
s2t2	-	s2t2
-	-	s3t3
s1t1	-	s4t4
s2t2	-	
s3t3	-	
s4t4	s4t4	

ShiftColumn = 2
ShiftLength = 3
(e)

s0t0	s0t0	s0t0
s1t1	s1t1	s1t1
s2t2	s2t2	s2t2
s3t3	s3t3	-
s1t1	-	s1t1
s2t2	-	s2t2
-	-	s3t3
s1t1	-	-
s2t2	-	-
s3t3	-	-
s4t4	s4t4	s4t4

Aligned Matrix
(f)

Figure 3: Example of Matrix Alignment (Source: [2])

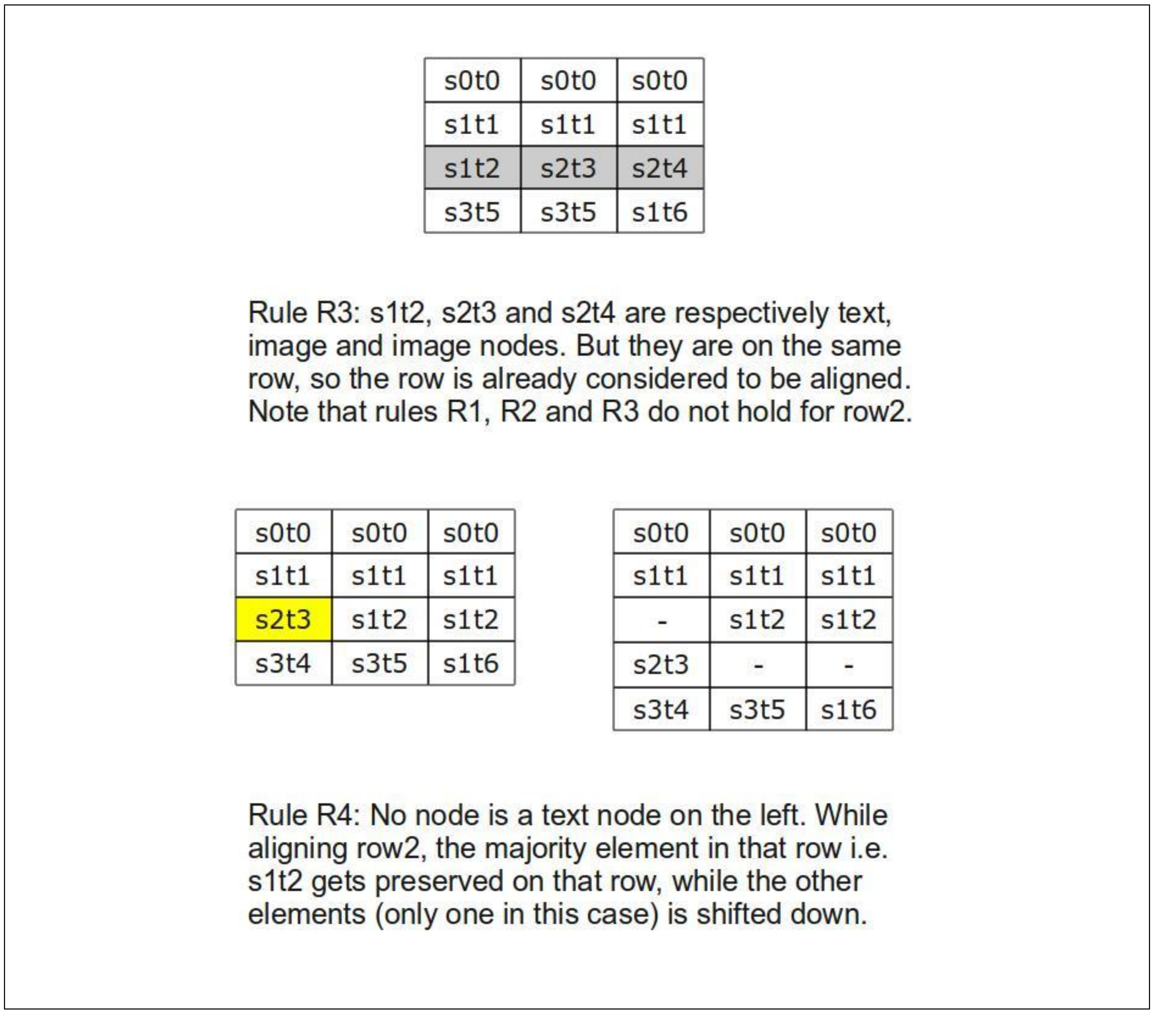
6.1 Alignment examples

6.1.1 R1 and R2

Figure 3 shows an example of Matrix alignment. This example is taken from [2]. We have replaced the symbols by our own $\mathbf{S}_k\mathbf{T}_l$ symbols. Six iterations are needed to complete the alignment. As we can see in Figure 3 (a), rows 0,1 and 2 are aligned. Row 3 is not aligned. We note that $\mathbf{S}_1\mathbf{T}_1$ in row 3 satisfies R1 (beccause span($\mathbf{S}_1\mathbf{T}_1$) = 2), hence we shift the column 2 down by one row, starting at $\mathbf{S}_1\mathbf{T}_1$. That aligns row 3. The algorithm proceeds to row 4 (Figure 3 (b)). $\mathbf{S}_4\mathbf{T}_4$ now satisfies R2, and so it gets shifted straight down to row 7. This results in the aligning of row 4. In this way, each individual row gets aligned, until the entire matrix is aligned, as in Figure 3 (f).

6.1.2 R3 and R4

Figure 4 shows the use of rules R3 and R4. Descriptions are provided, and the figure is self explanatory.



s0t0	s0t0	s0t0
s1t1	s1t1	s1t1
s1t2	s2t3	s2t4
s3t5	s3t5	s1t6

Rule R3: s1t2, s2t3 and s2t4 are respectively text, image and image nodes. But they are on the same row, so the row is already considered to be aligned. Note that rules R1, R2 and R3 do not hold for row2.

s0t0	s0t0	s0t0
s1t1	s1t1	s1t1
s2t3	s1t2	s1t2
s3t4	s3t5	s1t6

s0t0	s0t0	s0t0
s1t1	s1t1	s1t1
-	s1t2	s1t2
s2t3	-	-
s3t4	s3t5	s1t6

Rule R4: No node is a text node on the left. While aligning row2, the majority element in that row i.e. s1t2 gets preserved on that row, while the other elements (only one in this case) is shifted down.

Figure 4: Example of Matrix Alignment

The function *compressMatrix* compresses the matrix. For details, refer Section 6.
Once M has been compressed, we need to mark the variant data nodes. The function *listAllVariantNodes* does this. It marks two kinds of rows variant. The first are those that have different symbols but are data nodes. The second are those that are data nodes, and were merged by the compression

algorithm, and are also data nodes. Note that these will have a symbol of the form $\mathbf{S}_k\mathbf{T}*$, as described in Section 6. The list *bNodes* is used to store these rows. After this, *getAlignmentResult* picks out the aligning symbol of each row, and in case the row is in *bNodes*, it gets a special symbol b (to denote basic data node). This new list, called *childList* is returned.

7 Matrix Compression

The intuition behind Matrix Compression comes from the following example. Consider a section of a page where in the variant data appears embedded in a division. The Aligned matrix of the relevant section is shown in Figure 5. Clearly, such cases occur when objects having similar structure get shifted to lower rows because of textual disimilarity. In such cases we wish to merge these rows back and come up with a more general node holding the same structural information, but the text inside is now variant.

We use the following rules for compression -

1. If the row has no blank symbols the row is considered fixed and is skipped from compression.
2. If there exists a successive set of non-fixed rows having the same structure (i.e. having the same S_k), we merge the rows adhering to the following classifications
   (a) If the set of rows contain a column containing more than 1 non-blank symbol, we merge the rows into a node of set type.
   (b) If the set of rows contain a column which contains all # blank symbols we merge the rows into a node of **optional** type.
   (c) If both the above conditions are met the nodes are merged into a node A of **set** type which itself is a child of a new **optional** node B and B is returned.
   (d) The symbol of the node being returned is set as $\mathrm{S}_k\mathrm{T}_*$.

Examples of compression are shown in Figure 5. The figure is self explanatory in light of the above classifications. The algorithm given in Procedure 4 compresses the matrix, by first iteratively finding the sets of rows which need to be compressed and then calling the routine **CompressMatrix** which contracts the set of rows into a single row representing an object and returns the type deduced from the above given rules. The type is then handled by the routine **EnterNameinSymtab**.

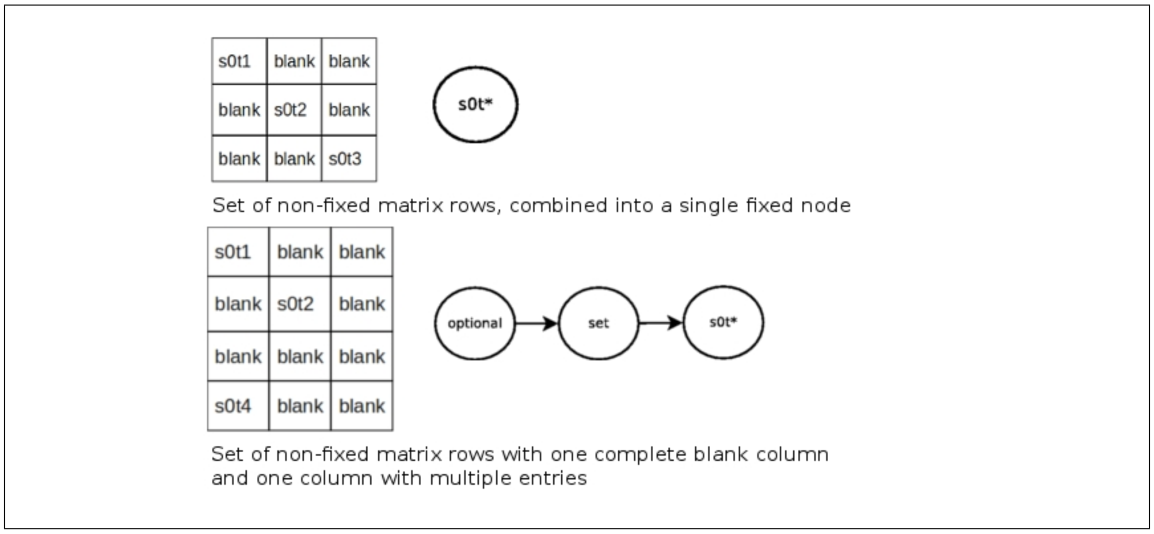

s0t1
blank
blank
blank
s0t2
blank
blank
blank
s0t3
s0t*
Set of non-fixed matrix rows, combined into a single fixed node
s0t1
blank
blank
blank
s0t2
blank
blank
blank
blank
s0t4
blank
blank
optional
set
s0t*
Set of non-fixed matrix rows with one complete blank column
and one column with multiple entries

Figure 5: Example of Matrix compression

Procedure 4 Compress the given matrix

```
Input: $M \leftarrow PeerMatrix$,$S \leftarrow SymbolTable$
Output: $M \leftarrow compressedMatrix$
  runningCompresion ⇐ false
  compressionStructure ⇐ null
  compressionStart ⇐ -1
  for $i0 \Rightarrow numberofRows(M)$ do
    rowsym ⇐ getNonBlankSymbolinRow(i)
    if $allBlankRow(i)$ then
      if $runningCompression$ then
        type ⇐ CompressMatrix(conpressionStart,i)
        EnterintoSymtab((CompressionStructure+t*),type)
        runningCompression ⇐ false
      end if
    else
      if $FixedRow(i)$ then
        if $runningCompression$ then
          if $CompressionStructure = ExtractStructure(SymbolinRow(i))$ then
            continue
          else
            type ⇐ CompressMatrix(conpressionStart,i)
            EnterintoSymtab((CompressionStructure+t*),type)
            CompressionStart ⇐ i
            runningCompression ⇐ true
            CompressionStructure ⇐ ExtractStructure(SymbolinRow(i))
          end if
        else
          CompressionStart ⇐ i
          runningCompression ⇐ true
          CompressionStructure ⇐ ExtractStructure(SymbolinRow(i))
        end if
      else
        if $runningCompression$ then
          type ⇐ CompressMatrix(conpressionStart,i)
          EnterintoSymtab((CompressionStructure+t*),type)
          runningCompression ⇐ false
        end if
      end if
    end if
  end for
  if runningCompression then
    type ⇐ CompressMatrix(conpressionStart,i)
    EnterintoSymtab((CompressionStructure+t*),type)
    runningCompression ⇐ false
  end if
```

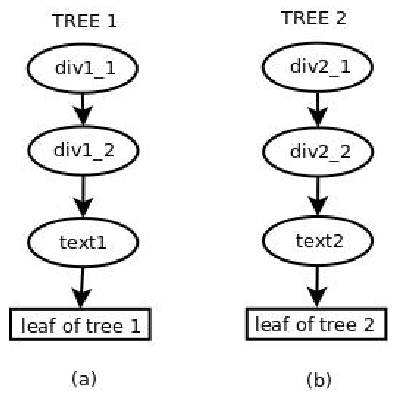


Figure 6: Need for structural and textual similarity

Why structural and textual similarities are needed

We would like to elaborate in some detail why we considered structural and textual similarities differently. This, along with matrix compression, exactly carries out the effect that we want to achieve. As an example, we consider the two DOM Trees in Figure 6 (In the figure, the numbers appended to **div** and **text** are only for disambiguation, and not a part of the node symbol).
What happens when there is no distinction between structural and textual similarity? Suppose that as the algorithm was descending into lower children, the divisions div1_1 and div2_1 got assigned the same symbol at that level (this happened because although the text they contain differs, they are large and have pretty much the same structure). But on the next level, their children div1_2 and div2_2 are small, and have a difference right at the first child. So, the matching score of div1_2 and div2_2 doesn't surpass the threshold, and the two text nodes are treated differently, and will probably end up as optionals (keeping the threshold low has other problems). This is wrong on two counts. First, the pattern overestimates the trees, and second, it fails to deduce that the data in div1_2 and div2_2 is actually variant.
Having structural and textual similarities, and matrix compression solves this problem. Even when the two text nodes get treated differently during alignment, they get compressed into one node during compression, and their data detected as variant.

8 Pattern Retrieval

At this point where in we have the childList from the Matrix Alignment algorithm, we move onto the problem of detection of repetitive data/pattern and coalescing them into a set type nodes. Essentially the problem is to be able to build a minimal regular expression covering the childList. The regular expression that we require here is of a weaker type, i.e. it contains only concatenation and + operators.

To this effect we introduce a basic structure called a **PatternNode** which has two attributes, **type**, which tells whether the pattern enclosed below the node is repeated or not and **children** which is a list of further patternNodes. Clearly the list of children is a reoresentative of the concatenation operator and the type encodes information about the + operator.

Hence the problem is given a list of successive symbols produce a list of patternNodes which represent the minimal regular expression. We outline the algorithm used in Procedure 5.

The algorithm takes in as input a list of symbols. Further it proceeds by detecting and merging continuous patterns of increasing size, i.e. in its first iteration the algorithm looks for patterns of size 1 and merges them together moving onto higher sizes.

To detect continuous patterns of a specific size, the algorithm iterates through all positions (pos) in the present pattern and tries to find a repetition of a pattern. In order to detect a continuous repetition of a pattern of size k at a position say s, the algorithm builds two regular expressions one(regexp1) representing the pattern from pos ⇒ pos+k and the other(regexp2) representing the pattern from pos+k ⇒ pos+2k. If either of the two patterns are contained (contained means the language represented by the one of the regular expression is a subset of the other)inside the other, the algorithm proceeds by removing the container and contained pattern(regular expression) and inserting a new PatternNode representing a set(repetitive pattern) of container pattern.

As an example consider the string "abbabcabcddd". The list of patterns(in terms of the regular expression they represent) after each iteration over possible patternLength is enumerated below -

1. Iteration 1 ⇒ abbabcabcddd
2. Iteration 2 ⇒ a(b)+abcabc(d+)
3. Iteration 3 ⇒ (a(b)+)+cabc(d+)
4. Iteration 4 ⇒ (a(b)+c)+(d)+)

The pictorial represention of the list of pattern nodes during each iteration is shown in Figure 7.

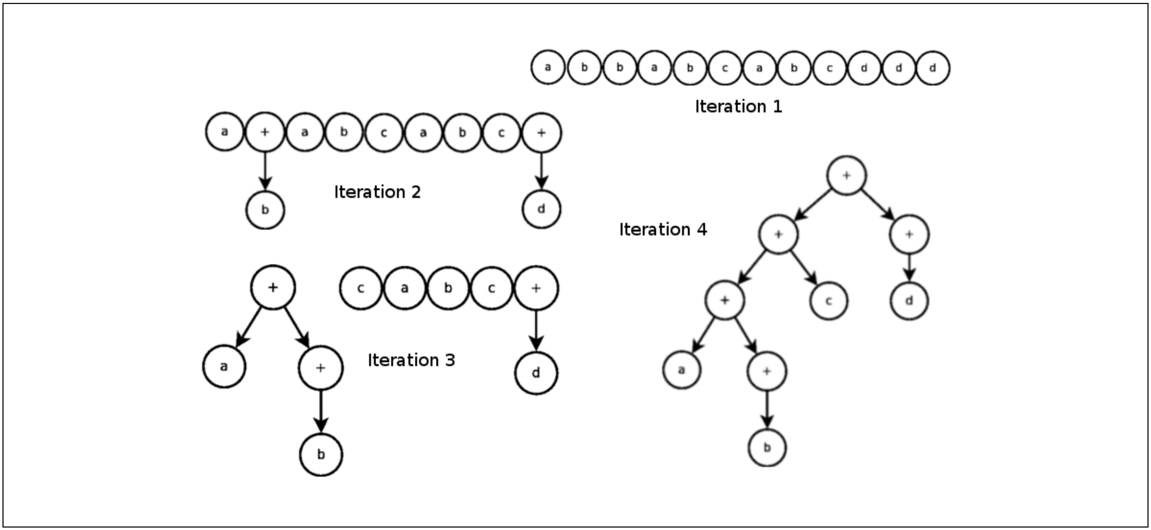


Figure 7: Iterations of the Pattern Mining Algortihm

A note on our pattern mining algorithm

The Pattern Mining algorithm outlined in [2], can go on to over-estimate patterns at times. For example, consider a string *abcbcbcdabcdcdcd*. The algorithm mentioned in the paper would convert the string into the following pattern - (*abcd*)+. A look into why it would do so is because the algorithm essentially looks for consecutive similar looking strings and once such repetitions of the same pattern is found it goes on to simply retain one instance of the pattern, which results in a loss of information. In the given example when the algorithm first identifies the set of consecutive $bc's$ and $cd's$, it simply retains one *bc* and *cd* and deletes the remaining occurences. This leaves the string as *abcdabcd*, from which it would go onto detect (*abcd*)+, which is clearly an overestimate of the pattern. The problem here is that while simply deleting the remaining occurences of *bc* or *cd*, we lose the information that it is a set and not one single occurence of the pattern. This is what we aim to solve by incorporating regular expressions into our pattern mining procedure. In our method, since the matching is done by way of containment of regular expressions, our method would first convert the above string to $a(bc)+dab(cd)+$, and then because neither pattern starting at the $a's$ i.e. $a(bc)+d$ and $ab(cd)+$ contains the other, the algorithm gives out the resulting pattern as $a(bc)+dab(cd)+$ which seems a better estimate. This problem of over-simplification is what we want to tackle with our pattern mining approach.

Procedure 5 Given a sequence of symbols generate a minimal pattern

```
Input: $l \leftarrow List < String >$
Output: $pattern \leftarrow List < PatternNodes >$
  for $i = 1 \rightarrow |l|$ do
    $pattern[i] \Leftarrow PatternNode(l[i])$
  end for
  $patternLength \Leftarrow 1$
  while $patternLength \leq |l|/2$ do
    for $pos = 0 \rightarrow |pattern|$ do
      $end = getPatLengthEnd(pattern, pos, patternLength)$
      if $end = -1$ then
        $pos \Leftarrow pos + 1$
        $continue$
      else
        $nextend \Leftarrow getPatLengthEnd(pattern, end, patternLength)$
        if $nextend = -1$ then
          $pos \Leftarrow pos + 1$
          $continue$
        else
          $regexp1 \Leftarrow buildRegExp(pattern, pos, end)$
          $regexp2 \Leftarrow buildRegExp(pattern, end, nextend)$
          if $compareRegexContainment(regexp1, regexp2)$ then
            $modifyPattern(pattern, pos, nextend, nextend, end)$
          else if $compareRegexContainment(regexp2, regexp1)$ then
            $modifyPattern(pattern, nextend, end, pos, nextend)$
          else
            $pos \Leftarrow pos + 1$
            continue
          end if
        end if
      end if
    end for
    $patternLength \Leftarrow patternLength + 1$
  end while
  return pattern
```

9 Data Extraction

Once the PatternTree P has been constructed from the input seed HTML trees, we move on to the problem using this recognized template. Here in we describe an algorithm that traverses the tree from top to bottom in both the HTML trees and the Pattern Trees simultaneously, matching both at each level and then descending down recursively. The algorithm tells whether the given HTML tree belongs to the language of P or not and in case it does it pulls out the text fields of all matched variant nodes and returns them in a list.

The procedure $MatchPatNDomTree$ to do so is described in Procedure 6. The algorithm tries to match the inital few elements of the input DOM list greedily according to the type of the pattern. The different cases and their outputs are outlined in Procedure 6. The output is a 3-tuple containing the matchResult, remaining unmatched list and extracted varaiant data.

To descend one level down the tree the function **runOnChildren** is called which is passed which takes as arguement a list of HTML trees and a list of Pattern Trees(the childlist of the Parent). The function runOnChildren iterates over the Pattern Tree trying to match it with the remaining list of HTML nodes(using **MatchPatnDomTree**, thereby implementing the recursion in two levels). With each succeding match we append the returned list of Variant Data. The optional class is handled here. In case during iteration we come across an Optional Node, we first assume that the optional node is present and try and match it. Whenever any node fails to match, we iterate back to the last optional that we assumed was present and then further re-run the iteration from that point, this time assuming that the optional node is not taken. In case there is no previously taken optional remaining the function returns a false match signalling that the node failed to match. This way we implement a full backtrack with respect to the taking of optionals.

The function $MatchPatnDomTree$ is initially called with a single HTML Tree(the page to be matched) and the PatternTree constructed by methods described in orevious sections and finally the returned list of Variant Data is stored.

10 Experiments

10.1 Strategy

The system described above was tested in the following manner (Note that due to time constraints tests were not done in an exhaustive manner but rather simple empirical tests have been performed. We wish to do a more robust test during the summer.)

1. We run the test to deduce the structures of 4 sites namely,
   (a) `www.xkcd.com`
   (b) `www.9gag.com`
   (c) `www.explosm.net`
   (d) `www.qwantz.com`
2. Each of the above site was downloaded and the HTML pages were extracted in a single folder. Further extraneous data such as Comments, DOCTYPE nodes and scripts were removed from the pages.
3. A certain number of pages say k(a parameter that we would like to further experiment on) was selected randomly from the pool of pages of a website and the Tree Merging Algorithm described in Section 3 was run on the k pages to extract one common template(patternTree) pervading the pages.
4. Given the patternTree extracted from the pages, the Data Extraction mechanism described in Section 8 is run on all the pages from the site, continually dumping the data extracted

Procedure 6 MatchPatNDomTree - match the pattern tree with the starting elements of the list of DOMTrees

```
Input: l ← list<HTMLTree>, p ← patternTree
Output: <status,remainingList,variantList>
  if p.type is single then
    if root(l[0]) equals p.symbol then
      return runOnChildren(children(l[0]),p.childList)
    else
      return <-1,l,{} >
    end if
  else if p.type equals variant then
    if isDataType(symbol(l[0])) then
      return <1,subList(l,1...end),{ExtractText(l[0])} >
    else
      return <-1,l,{} >
    end if
  else if p.type equals set then
    run ← runOnChildren(children(l),p.childList)
    if run was unsuccessful then
      return <-1,l,{} >
    else
      while run is successful do
        run ← runOnChildren(remaining(run),p.childList)
        append run.variantList → globalVariantList
      end while
      return <1,remaining(run),globalVariantList>
    end if
  else
    Error
  end if
```

from the page(if the page matches the structure) into a Lucene[1] Index linking the data with the page URL. In case the page doesnt match the PatternTree, the page is recorded as a non-matching page for later analysis.

10.2 Empirical Results

The following data have been collected with a 2.5% learning ratio, i.e., the pattern is created on 2.5% of the total files, or 20 files, whichever is greater (note that files are randomly chosen). The following table highlights results of some elementary tests that we performed. The numbers shown denote the number of files to match the pattern. The total number of files for that website is denoted in brackets in the first row.

	xkcd (880)	Dinosaur (1960)	Cyanide and Happiness (133)	9gag (139)
s^*=0.80, $t^\dagger$=0.90	39	300	18	48
s=0.90, t=0.95	39	400	18	70
s=0.70, t=0.75	39	366	18	25

$*$ s stands for structureDelta
† t stands for textDelta

We note that while xkcd and Cyanide and Happiness have the same success rate irrespective of the variation in the parameters, Dinosaur seems to work best for average values of parameters, whereas 9gag works best for low values. But, the number of unmatched pages is not the only parameter that should be looked at. The amount of variant data retrieved is important. Also, fixed text should not be retrieved. We plan to perform these experiments in the future (see Section 10).

11 Future work

While the current system is working well, and extracts variant data from websites satisfactorily, there are still a few things that need to be done. We plan to implement them in due course of time, and outline them here.

1. **Making data extraction more robust:** The current data extraction algorithm, outlined in Procedure 6 has a basic flaw. In case some subtree at some depth does not correctly match and fit into the pattern, the error is propagated up, and the whole tree gives a failed message. This happens even if the rest of the tree properly fit into the pattern. The problem with this is that the algorithm can't overlook minor errors, and therefore, misses out on a lot of relevant data.

2. **Rigorous testing:** To compare the performance of our implementation to that of existing ones, we need to test ours on a benchmark, and then compare results. Tests are outlined in [2].

3. **Analyse effect of parameters:** The parameters *structureDelta* and *textDelta* are crucial. They are the thresholds for considering two trees same or not same. Low values will make us create structures with unnecessarily merged trees, which might not match DOM Trees during data extraction. High values will overestimate the pattern, and probably create too much variant data, which is undesirable. Currently, we have empirically fixed values, but it is essential to study how they affect patterns, and the extracted data. Detailed tests need to be done for it.

References

[1] Lucene indexing tool, http://lucene.apache.org.

[2] Chang, Kayed, Shaalan, and Girgis. Fivatech:page-level web data extraction from template pages.

[3] Yang, W. Identifying syntactic differences between two programs.