Knowledge and Information Systems, September 2008, Volume 16, Issue 3, pp 303-329. The final publication is available at at Springer via http://dx.doi.org/10.1007/s10115-007-0108-0.

Under consideration for publication in Knowledge and Information Systems

S2S: Structural-to-Syntactic Matching Similar Documents

Ramazan S. Aygün

Computer Science Department, University of Alabama in Huntsville, Huntsville, AL 35899, USA

Abstract. Management of large collection of replicated and versions of data in centralized or distributed environments is important for many systems that provide data mining, mirroring, storage, and content distribution. In its simplest form, the documents are generated, duplicated and updated by emails and web pages. Although redundancy may increase the reliability at a level, uncontrolled redundancy aggravates the retrieval performance and might be useless if the returned documents are obsolete. Document similarity matching algorithms do not provide the information on the differences of documents, and file synchronization algorithms are usually inefficient and ignore the structural and syntactic organization of documents. In this paper, we propose the *S2S* matching approach. The *S2S* matching is composed of structural and syntactic phases to compare documents. Firstly, in the structural phase, documents are decomposed into components by its syntax and compared at the coarse level. The structural mapping processes the decomposed documents based on its syntax without actually mapping at the word level. The structural mapping can be applied in a hierarchical way based on the structural organization of a document. Secondly, the syntactic matching algorithm uses a heuristic look-ahead algorithm for matching consecutive tokens with a verification patch. Our two-phase *S2S* matching approach provides faster results than currently available string matching algorithms.

Keywords: String matching; Information retrieval

1. Introduction

With the progress of technology and the cost reduction of storage, update, and transmission of data, the information is available in lots and various ways to the public. People can subscribe to web pages, newsgroups, and email groups at their will with the anticipation of attractive information. If someone uses "google" to

Received xxx
Revised xxx
Accepted xxx

search data on the web, it is likely to get almost identical documents for a set of keywords. Information retrieval systems are eager to return all the duplicates and versions of documents whether they are obsolete or not. Nowadays, the number of available documents to the information retrieval (IR) systems has been enormous. It is left to the user to identify and filter the necessary information.

1.1. Motivation

The amount of digitally available information has increased with the support of database systems, information retrieval systems, web servers, and email servers. The information can be copied, updated, and transmitted to different resources. Although the systems can synchronize with each other (i.e., the files at different systems are synchronized), the meaningful difference between documents are important for users to make critical decisions. We are particularly interested in the following applications:

a) **Email and Newsgroups**. The users may receive duplicates of messages when there are network errors and the sender sends the same message more than once unintentionally. If the sender forgets some important information in the earlier message, the message is submitted with minor differences. The updates may be a couple of words like apartment number and/or a zip code in an address, and date for a deadline. For example, you want to retrieve the messages for Mobile Database conferences. The system returns all the call for papers for the conferences with updated deadlines. The sender of the message sometimes identifies the difference with the previous message in the latest message. Sometimes, it is not clear whether the message is new or an updated version of the older message. We expect the system to disclose what has been new in the latest message. The user should not go over two messages and compare them to identify the updates if the differences are not stated explicitly. When the newer message contains all the information of the previous message, the older message becomes obsolete and that message can be deleted from the database.

b) **Replication in P2P Systems**. The peers in P2P systems (Milojicic et al, 2002) can search data at different peers, and then store, update, and share with other peers. As new peers are added, data are replicated and updated; and the queries will yield the output of similar documents. Since most peers connect to the network using slow bandwidths, a good summary of differences between two documents will help the user access the desired document.

c) **Data Mining and Information Retrieval**. The performance of data mining and information retrieval systems is directly influenced with the amount of data to be processed. If the duplicate, similar, and obsolete data are identified, the retrieval process will be faster and data mining will result more accurate results.

d) **Subscription systems.** The browsers such as Internet Explorer allow users to subscribe web pages for periodical storage and update of locally stored web pages. Updating the data efficiently is useful, but it would also be useful to provide the syntactic differences between the older and newer versions rather than comparing them manually.

e) **Web crawling.** There has been tremendous effort to synchronize large sets of web pages and this has yielded to web re-crawling research (Brewington et al, 2000; Cho et al, 2000). If the source to be synchronized is not known or if

there are multiple resources, it is important to know the site (or document) to synchronize with. A syntactic difference of documents would be a good indicator.

The common theme in these applications is the presentation of differences among similar data before proceeding to update, copy, or delete data. Our goal is to provide an efficient preliminary method to present the differences among similar documents.

1.2. Related Work

The string processing research area is a well-studied and analyzed research (Aho et al, 1976; Amir et al, 2004; Apostolico, 1996; Hirscberg et al, 1977; Nakatsu et al, 1982; Navarro, 2001; Chen et al, 2006). The problem that we investigate is close to finding the longest common subsequences between two strings S and T. The longest common subsequence is a state of the art similarity measure for sequences, and is widely used in sequence related tasks (Wang et al, 2006). The traditional string processing algorithms do not assume anything on the similarity of strings. In other words, these algorithms have to consider worst cases where the input strings are dissimilar.

The dynamic time warping is a dynamic programming approach to determine the similarity between two time series (Sankoff, 1983). The typical distance between two time series is determined by the following formula:

$$D_{i,j} = min(D_{i-1,j}, D_{i-1,j-1}, D_{i,j-1}) + d(s_i, p_i);$$

where function d is a distance function that depends on the application for time series S and P; and D is a distance matrix to be used for dynamic programming. In this paper, our method for structural mapping is also a kind of dynamic time warping algorithm. However, the initial conditions and the distance functions are generated to serve our purposes.

The document similarity methods usually return the similarity of two documents. However, they do not return any information on what is similar and what is not. String distances are usually expensive to compute in large databases where each document may contain thousands of words. The most efficient document retrieval methods use Latent Semantic Indexing based on frequency matrices (Deerwester et al, 1990; Dumais, 1991). These methods provide fast document similarity measures. In (Broder, 1997), two measures on the resemblance and containment of documents are proposed that corresponds to "roughly the same" and "roughly contained". Since traditional string distance measures (Hamming, Levenstein, etc.) are usually expensive to compute, they use these new measures based on shingles (the length of consecutive tokens). However, their algorithm does not state the actual differences between documents.

The granularity of alphabets affects the performance of the algorithms. Although text documents are composed of characters, characters are not adequate to evaluate to obtain meaningful differences between two strings. The *diff* application that compares two documents using an efficient longest common subsequence algorithm is an example state-of-art technique for text differencing (Schubert et al, 2005). The granularity of *diff* command in UNIX is a line. Each line is treated as a token (alphabet). Even a word is shifted in each line to the next line, the documents will be totally treated as different documents. The *diff*

command aims to reduce the number of changes to convert one document to another. There are also other comparison algorithms like *bdiff* and *vcdiff* (Hunt et al, 1998; Korn et al, 2002). The algorithms are usually studied under delta compression techniques (Hunt et al, 1998; Korn et al, 2002; Miller et al, 1985). The delta coding is used to convert one file to another based on differences and distances between two files.

The *rsync* algorithm (Trigdell, 2000; Trigdell et al, 1996) is commonly used to synchronize files. It serves to synchronize files that may exist in any format by dividing first into blocks and then generating hash keys. The hash keys are matched in the new file to synchronize two files. The update information is reported in terms of blocks which may not carry semantic information for meaningful documents. The rsync algorithm focuses on the reduction of the number of bits transferred and it does not have any good performance bounds with respect to edit distance (Savant et al, 2003). Even if there is a single mismatch within a block, that is considered as a different block.

1.3. Our Approach

In this paper, we propose the *S2S* matching that is composed of structural and syntactic evaluations to compare the similar documents. Our goal is not to give the longest common subsequence that returns the minimum editing distance. Our goal is also not to convert a document to another efficiently. Our goal is to return the meaningful differences between similar documents that can be used in information filtering.

In our algorithm, we do not treat every appearance of a member of the alphabet in the same way. We assume that the strings have structure and closeness of appearances (or existence of an appearance in the same component) may affect the matching process. Since string matching is a costly process, we firstly determine the similar documents by using document similarity measures and then apply our *S2S* matching algorithm. Our syntactic evaluation algorithm uses a look-ahead heuristics for 2 consecutive matches. The document is virtually divided into components (sentences in text documents) and the appearance of an alphabet (e.g. words) in different components (e.g. sentences) can be considered as different. Fig. 1 depicts the steps of our approach. We have performed our experiments on text documents.

Our contributions can be listed as follows:

- *S2S:* Two-phase matching process is composed of structural mapping and syntactic evaluations.
- Structural mapping algorithm is optimized and has $O(n)$ complexity.
- Syntactic matching avoids the incorrect matching of documents in different components and also has $O(n)$ complexity for similar documents.

Since our algorithm is a heuristic algorithm, there might be cases where the output is not semantically correct. In those cases a worst-case algorithm should process the original data or another efficient algorithm including this one can reevaluate the generated output. But this is not discussed in this paper. We make sure that the complexity of the algorithm is $O(n)$ by considering similar documents, so it will not deteriorate the overall complexity. This paper is organized as follows. The following section explains the background on string

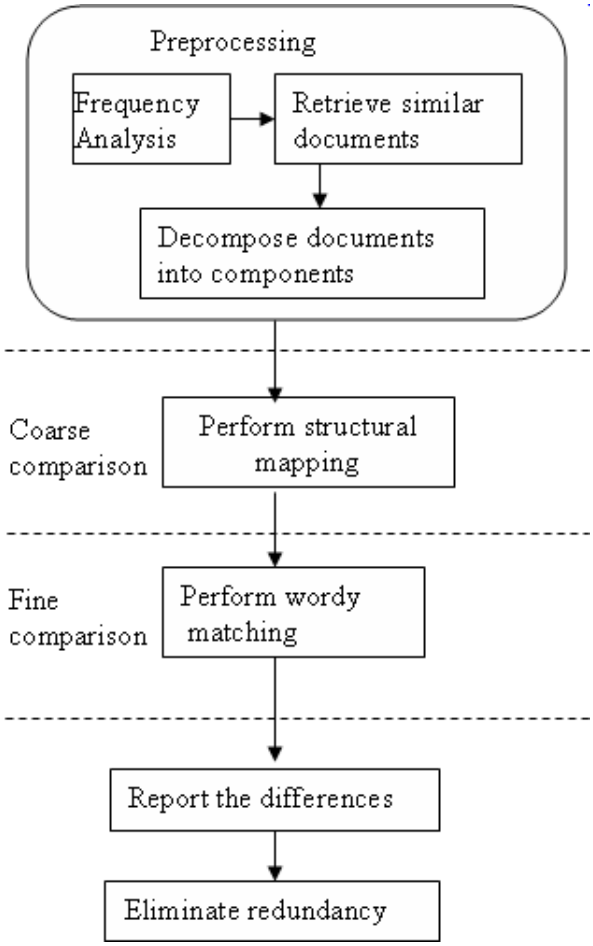


Fig. 1. Steps of structural and syntactic document matching

matching. Our *S2S* approach is summarized in Section 3. Structural evaluation is discussed in Section 4. Syntactic evaluation is explained in Section 5. Section 6 discusses our experiments and gives analysis of our algorithm. Section 7 discusses limitations and future work. The last section concludes our paper.

2. Background on String Matching and File Synchronization

Let $S = s_0 s_1 ... s_{n-1}$ and $T = t_0 t_1 ... t_{m-1}$ be two strings defined over alphabet Σ. The length of S is denoted with $|S| = n$; s_i denotes the i^{th} alphabet of S; and $s_{a..b} = s_a s_{a+1} ... s_b$ where $0 \leq a < b \leq n$. We consider three edit operations: deletion, insertion, and substitution. In this paper, when comparing S and T, if a substring of S does not appear in T, it is treated as deletion. If a substring of T does not appear in S, it is treated as insertion. Fig. 2 shows a sample sequence

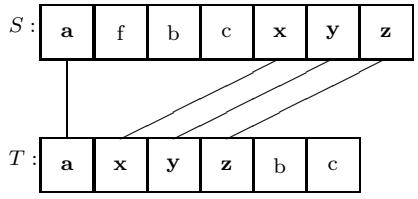


Fig. 2. Sequence matching.

matching. The longest common subsequence (LCS) is *axyz* for this example. If we want to convert S to T, remove *fbc* from S and substitute (second) a with *bc*. To convert T to S, we would insert *fbc* between a and x and substitute *bc* with a. In terms of the number of operations, only two operations are needed to convert S to T and T to S. The edit distance between S and T is 5 (e.g, insert *fbc*; substitute a with c; and insert c). The *dist(S,T)* gives the edit distance between S and T.

The granularity of alphabet is important in assessment of similarities and differences between two strings. If the semantics is not important and if the consideration is to minimize the number of operations to convert a text to another, then the alphabet may consist of lines in text documents as in *diff* program. This will lead to a fast comparison but incorrect interpretation of differences between strings. For example, consider the documents in Fig. 3. The *diff* algorithm returns the following delta encoding: $\{\{1c1, 3\}, \{3c5\}, \{5c7, 15\}$. That is, substitute line 1 of S with lines of 1 to 3 of T; substitute line 3 of S with line 5 of T; and substitute line 5 of S with lines 7 through 15. The actual semantically correct answer should be to insert first five lines of T into the beginning of S. If the granularity of the alphabet is a character, the program may even consider an extra space or a tab as a mismatch although the rest of the text is the same. This is also an issue with the rysnc (Trigdell, 2000; Trigdell et al, 1996) algorithm.

In our approach, we consider the granularity of an alphabet as a word. From now on, an alphabet corresponds to a word. Throughout paper, we also use the word token to represent the members of Σ.

3. S2S

The *S2S* matching approach compares documents in two phases: structural and syntactic. The structural mapping considers the structural organization of documents and does not look into semantics. On the other hand, syntactic evaluation considers matching of the alphabet. In our system, the syntactic matching follows structural mapping. The structural mapping identifies the set of components of the documents where syntactic matching needs to be performed. Since these components are likely to have some similarity, the syntactic matching can be performed very fast.

The structural mapping does not filter out any syntactic meaning. It is just a preprocessing step to map components for syntactic matching. It just reduces the burden on the syntactic matching. For example, assume that there are two versions of a paper: p_1 and p_2. We would like to see how p_2 is updated from p_1. In a traditional approach, the complete document is treated as a single string and

```
1 ====================================================================
2 Apologies if you receive multiple copies of this message.
3 ====================================================================
4 Due to several requests, the submission deadline has been extended
5 NEW DEADLINE: June 10th, 2004
6 ====================================================================
7 Call for Papers for a Special Issue in
8 International Journal of Comp. Systems Science and Eng. IJCSSE
9 Special Issue on Mobile Systems ...
```

(a)

```
1 Call for Papers for a Special Issue in
2 International Journal of Comp. Systems Science and Eng. IJCSSE
3 Special Issue on Mobile Systems ...
```

(b)

Fig. 3. (a) new document, T (b) old document, S

then compared. However, it is not necessary to compare a word in the abstract with a word in the conclusion of the paper. Our structural mapping indicates that abstract of p_1 should be compared against the abstract of p_1 and the conclusion of p_1 should be compared against the conclusion of p_2. However, if the documents are similar to each other and moreover, if all the words that are searched exist at a relatively close distance from the beginning of the search point, hierarchical decomposition may not affect (or improve) the performance. For example, if two documents are exactly the same, structural mapping will not improve the performance. However, such cases cannot be determined in advance, and structural mapping improves the performance especially when new words, sentences, and sections are introduced in a new document. If the syntactic mapping correspond to matching words, word level matching is performed for the corresponding sentences only. Therefore, a word is not searched beyond the end of corresponding sentences.

3.1. Structural Mapping

Since the text documents are not structured as XML or web documents, the document similarity methods such as (Hammouda et al, 2004; Li et al, 2005) cannot be applied to plain text documents. However, documents can still be divided into smaller components. For a regular document at the high level, a document is composed of sections and sections are composed of subsections. The subsections are composed of paragraphs and paragraphs are composed of sentences. Sentences are composed of words and words are composed of characters. The lowest level for structural mapping is the level of sentences. The actual similarity of sentences can be accomplished by comparing the words in two sentences.

The identification of the lowest level is determined by syntactic matching. For example, if the syntactic equivalence of two paragraphs can be identified by just comparing the sentences (but without looking into words in the sentences),

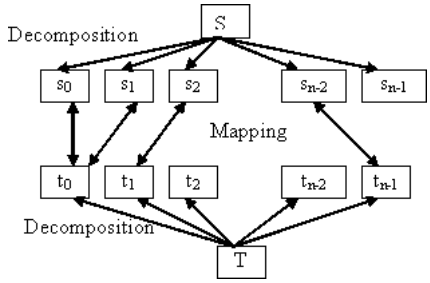


Fig. 4. Structural mappings of documents S and T

the lowest level for structural mapping is a paragraph. The number of levels of decomposition of a document depends on the type of a document. The structural mapping can also be performed at multiple levels in a hierarchical way.

It should be noted that the identification of structural organization of a document is domain dependent. The domain of a document might be a book, report, journal (or conference) paper, survey, and test. Some documents may use different fonts to identify section headings while some others might use numbering for section titles. However, sentence and word level operations are not domain dependent. Starting from the sentence level is applicable to all domains.

The matching always starts from the structuring mapping and then syntactic mapping follows structural mapping. The structural mapping might have different levels such as sections, paragraphs, and sentences. The structural mapping also starts from the highest level possible. If section and sentence are two levels available for structural mapping, the structural mapping starts from section level mapping followed by the sentence level.

In our case, the documents are decomposed into components (in our case, sentences) by using delimiters such as ".", "?", and ";" symbols. For each string S, we keep an array for delimiter characters. The structural mapping requires the mapping of components at the structural level without considering any semantics. If the mapping levels are sentences, the sentences of two documents are mapped. Fig. 4 depicts the structural mapping of two documents.

The mapping is not always $1-1$ mapping. Sometimes, there is no correspondence and this corresponds to insertion or deletion of a component. In some cases, component may be divided into components or a set of components is united as a single component. At the structural mapping level, it is not possible to judge on insertions or deletions. The structural mapping always assigns a corresponding component and it is the responsibility of syntactic matching to determine insertions and deletions.

3.2. Syntactic Matching

We realize two issues when we process syntactic matching. Firstly, some sets of characters like white space, tab, and extra lines do not affect the semantics of the document. Secondly, an occurrence of a word in another sentence cannot always be considered as a correct match. Especially consider articles *a* and *the*. An occurrence of *the* in another sentence is probably a different occurrence.

If a word, w, is appearing more than k sentences later where k is a threshold,

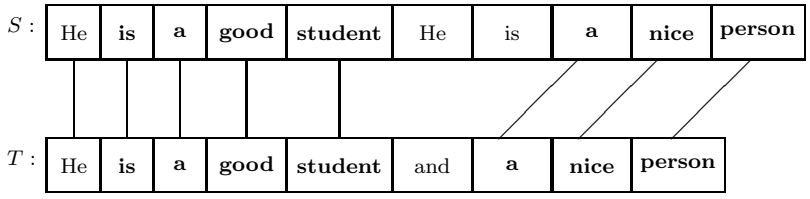


Fig. 5. Splitting of sentences.

that appearance can be treated as an incorrect match. Normally, it is reasonable to consider only matches within a sentence. However, the cases where a sentence is divided into multiple sentences or multiple sentences are united into a single sentence might occur and these cases might be missed. In the example shown in Fig. 5, the word nice in t is matching with the word nice in the second sentence of s. Although this match appears in different sentences, this is a correct match. We provide reliable algorithms in Section 4 and 5.

4. Structural Mapping

Assume the matching of two documents V and W where $|V| = n$ and $|W| = m$. If n and m are huge numbers, the matching with algorithms having $O(mn)$ complexity is costly. We will use the fact that all documents are structured. A document can be considered as a hierarchical organization of syntactic information. At the high level there are sections, subsections, and so on. We know not all documents are properly hierarchically organized. But we definitely know that all documents are composed of sentences. Structural mapping algorithm returns all the mappings to be evaluated by syntactic matching. Since the problem is divided into smaller problems after mapping, parallel processing algorithms can benefit the mappings of structural mapping.

Let $V = v_0v_1...v_{g-1}$ and $W = w_0w_1...w_h$ where v_i and w_i denote the i^{th} components of V and W, respectively (Note: when structural mapping is considered, all indices refer to components rather than alphabets). Let $|v_i|$ denote the length (the number of words in v_i). If $|v_i| = |w_j|$, we can state that the number of insertions and deletions is equal. The length of a component plays an important role whether they are similar or not. Depending on the length of components an approximate cost function can be determined as follows:

$$
\begin{aligned}
&C_{0,0} = 0\\
&C_{0,j} = |v_j| + C_{0,j-1} \; for \; 1 \leq j \leq n;\\
&C_{i,0} = |w_i| + C_{0,i-1} \; for \; 1 \leq i \leq m;\\
&C_{i,j} = min(C_{i-1,j}, C_{i-1,j-1}, C_{i,j-1}) + ||w_i| - |v_j||;
\end{aligned}
$$

where C is a cost matrix and $C_{i,j}$ is obtained by adding the minimum insertion/deletion distance for the i^{th} and j^{th} component. The minimum insertion/deletion distance is $minInsDelDist(v_i, w_j) = ||v_i| - |w_j||$. This algorithm forces matching components. In other words, when two components are compared, the distance is the difference in the number of words. This formula resembles dynamic time warping formulas. However, this type of application of

		51	27	15	10	14	18	9	26	16	19	1	19
	0	51	78	93	103	117	135	144	170	186	205	206	225
51	51	**0**	27	42	52	66	84	93	119	135	154	155	174
27	78	27	**0**	15	25	39	57	66	92	108	127	128	147
15	93	42	15	**0**	10	24	42	51	77	93	112	113	132
10	103	52	25	10	**0**	14	32	41	67	83	102	103	122
14	117	66	39	24	14	**0**	18	27	53	69	88	89	108
18	135	84	57	42	32	18	**0**	9	35	51	70	71	90
9	144	93	66	51	41	27	9	**0**	26	42	61	62	81
15	159	108	81	66	56	42	24	15	**11**	27	46	47	66
11	170	119	92	77	67	53	35	26	**22**	16	35	36	55
16	186	135	108	93	83	69	51	42	36	**22**	19	20	39
19	205	144	127	112	92	74	52	52	49	25	**22**	23	20
1	206	145	128	113	93	75	53	53	50	26	23	**22**	21
19	225	264	147	132	102	94	72	63	60	45	26	27	**22**

Table 1. The dynamic programming sample for computation of the cost matrix.

dynamic warping is new for string matching, since in string comparison, words are matched; not the length of the components.

If we consider the insertion or deletion of sentences in distance function, then it should be modified as follows:

$$C_{i,j} = min(C_{i-1,j}, C_{i-1,j-1}, C_{i,j-1}) + min(|w_i|, |v_j|, |w_i - v_j|);$$

Please note that initial formulation does not ignore insertions and deletions. Deletions and insertions are detected at the syntactic matching level. In the second formulation, it says that the distance between two sentences whose lengths are 5 and 40 is min(5,40,35)=5 that corresponds to deletion/insertion of the smaller sentence.

However, the previous formulation does not indicate the relationship between the previous cost value and the new additional cost. If the diagonal value is the minimum value, then the substitution is preferred. If the upper or left value is the minimum value, there is insertion/deletion. The previous equation is updated as follows:

$$C_{i,j} = min(C_{i-1,j} + |w_i|, C_{i-1,j-1} + |w_i - v_j|, C_{i,j-1} + |v_j|);$$

In our experiments, we have used the third equation.

Table 1 shows a portion of a sample cost matrix. The first row shows the length of sentences in W and the first column shows the length of sentences of V. After the cost matrix is generated, the matrix needs to be traced back to identify the components to be mapped. We use two one-dimensional arrays for each string. The bold values in Table 1 show the minimum values. Whenever a value is computed for a matrix location, we also maintain the previous location (i.e. diagonal, left, or upper) in the matrix that leads to this new value.

The *TraceMatch* algorithm given in Algorithm 4.1 starts from the bottom-right corner and traces for the minimum value in the direction of top-left corner. Since the size of the cost matrix is $(g+1, h+1)$, the indices of mapping sentences is 1 less than the indices of the cost matrix. The Z values keep the mapping components of two documents. In other words, the component $Z_1(i)$ of W is mapped to $Z_2(i)$ in V. Note that $Z(i) \geq Z(i-1)$. Table 2 shows the mapping for the cost matrix given in Table 1. Depending on this table, $\{v_0, w_0\}$, $\{v_1, w_1\}$, $\{v_2, w_2\}$, $\{v_3, w_3\}$, $\{v_4, w_4\}$, $\{v_5, w_5\}$, $\{v_6, w_6\}$, $\{v_9, w_9\}$, $\{v_{10}, w_{10}\}$, $\{v_{11}, w_{11}\}$,

Z_1:	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**7**	**8**	**9**	**10**	**11**
Z_2:	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**

Table 2. Mapping components.

		51	27	15	10	14	18	9	26	16	19	1	19
	0	51	78	93	103	117	135	144	170	186	205	206	225
51	51	**0**											
27	78		**0**										
15	93			**0**									
10	103				**0**								
14	117					**0**							
18	135						**0**						
9	144							**0**	26	42			
15	159							15	**11**	27			
11	170							26	**22**	16			
16	186							42	36	**22**			
19	205										**22**		
1	206											**22**	
19	225												**22**

Table 3. Optimized algorithm to compute cost matrix.

and $\{v_{12}, w_{12}\}$ can be considered as $1-1$ mapping. The v_7 is mapped to multiple components in W. This is a possible indication of addition of sentences or splitting of a component. In this case, we map $\{v_6 - v_8, w_6 - w_9\}$. The sentences at the boundaries must also be included in this mapping.

Algorithm 4.1. The implementation of $TraceMatch$ algorithm.

Procedure $TraceMatch(C, V, W)$
IN: C is the cost matrix.
IN: $V = v_0 v_1 ... v_{g-1}$ and $W = w_0 w_1 ... w_{h-1}$.
OUT: Mapping arrays Z_1 and Z_2.
begin
$count = 0$;
$i = g$;
$j = h$;
$Z_1[count] = i - 1$;
$Z_2[count] = j - 1$;
increment *count*;
while ($i > 1$ and $j > 1$) **do**
begin
find min neighbor of $C_{i,j}$
update (i, j) to the *min* neighbor index
$Z1[count] = i - 1$;
$Z2[count] = j - 1$;
increment *count*;
endwhile
if remaining unmapped sentences **then**
map these sentences to the 1^{st} sentence of the other document
endif
reverse $Z1$ and $Z2$
end

4.1. Optimization of Structural Mapping

The complexity of determining the mappings is $O(gh)$. If the time complexity is $O(n)$ for regular syntactic matching, the time complexity of *S2S* matching would be $O(gh + \alpha n)$ where α is the number of mappings (for Table 2, $\alpha = 12$). Although this is faster than matching word by word, the gh factor may lead to $O(g^2)$ factor if $g \approx h$. If this is the dominating factor, it will have a quadratic complexity. In Table 1, bottom-left and upper-right corners have the maximum values. Those corners state that there are no sentences that could be mapped. We do not need to compute every value of the cost matrix. Especially, if the number of subcomponents (e.g. sentences or words) matches exactly, there is no need to match these components against other components. The components may also match from the end of the document. The components that match from the end of the document is marked and our algorithm is forced to match those components matching exactly at the end.

Definition 4.1. $P(C, r, c)$ is called partial cost matrix of C where $0 \leq i \leq r$ and $0 \leq j \leq c$ and $P(C, r, c)_{i,j} = C_{i,j}$.

Assume that $P(C, r, c)$ is the best match for r and c sentences of V and W. We want to estimate $P(C, r + 1, c + 1)$. There are three options: a) match sentences v_{r+1} and w_{c+1} b) insert sentence of V c) delete sentence of V. The heuristics is as follows: if matching is correct, the minimum value will be at $C_{r+1,c+1}$; and $C_{r+1,c+1} < C_{r+1,c}$ and $C_{r+1,c+1} < C_{r,c+1}$. Otherwise there is no match for these sentences and one of these sentences should be ignored. We estimate the other cell and compute the matrix as follows:

$C_{0,0} = 0$
$C_{0,j} = |v_j| + C_{0,j-1}\ for\ 1 \leq j \leq n;$
$C_{i,0} = |w_i| + C_{0,i-1}\ for\ 1 \leq i \leq m;$
$C_{i,j} = min(C_{i-1,j} + |w_i|, C_{i-1,j-1} + |w_i - v_j|, C_{i,j-1} + |v_j|);$
$C(i-1, k)\ is\ not\ computed\ if\ C(i+1, j+1) == 0\ for\ j \leq k \leq n$
$C(k, j-1)\ is\ not\ computed\ if\ C(i+1, j+1) == 0\ for\ i \leq k \leq m$

The optimized algorithm is given in Algorithm 4.2. The computation of $P(C, r+1, c+1)$ is based on $P(C, r, c)$ and the assumption is that we do not need to compute each cell since the local minimum will be in the neighborhood of the last minimum cell of $P(C, r, c)$. The heuristics is to follow the local minimum. It is very likely that this local minimum will lead to global minimum. We explain the success of this local minima tracking in the experiments.

Algorithm 4.2. The algorithm for the optimized structural mapping.

Function *int* OptimizedStructuralMapping (V, W)
IN: V, W: input strings
OUT: C: cost array
begin
$matchcount = 0$;
$C_{0,0} = 0$;
for $i = 1$ to g **do**
$C_{i,0} = V_{i-1} + C_{i-1,0}$;
endfor
for $j = 1$ to h **do**
$C_{0,j} = W_{j-1} + C_{0,j-1}$;

```
    endfor
    i = 1; j = 1;
    while w_i == v_j do
      increment i and j;
    endwhile
    decrement i and j;
    s_i = i and s_j = j;
    i = g − 1; j = h − 1;
    while w_i == v_j do
      decrement i and j;
    endwhile
    increment i and j;
    e_i = i and e_j = j;
    i = s_i; j = s_j;
    while i < e_i and j < e_j do
      old_i = i;
      old_j = j;
      compute the values of the neighbor cells
      group_cells(C, V, W, m, n, i, j);
      find min neighbor of C_{i,j}
      update (i, j) to the min neighbor index
      if i = old_i + 1 and j = old_j + 1 then
        matchcount++;
      endif
      if i = 2 and j = 2 then
        matchcount++;
      endif
      if C_{i,j} is 0
        skip columns up to j;
      endif
    endwhile
    return matchcount;
  end
```

4.2. How to use structural mapping

Although heuristic matching produces fast results, in rare cases it may not yield correct results. This heuristics depends on the minimum insertion/deletion distance between components. It is good to have an indicator on how to use this heuristics. Standard deviation of length of sentences is a candidate for such an indicator

$$\sigma(V) = \sqrt{\frac{\Sigma_{i=0}^{n}(|v_i| - \mu(V))^2}{n+1}}$$

and

$$\mu(V) = \frac{\Sigma_{i=0}^{n}|v_i|}{n+1}$$

If $\sigma(V)$ is close to 0, structural mapping might fail. It means that any component can map any component.

Assume that we have three different granularities for a document: word, sentence, and section. If structural mapping for sections might fail, structural mapping is only applied at the sentence level. If structural mapping at the sentence level might fail, structural mapping is skipped and syntactic mapping is applied directly. In practice, most documents have components at varying lengths. We have not encountered a situation where all mappings would fail, because most documents have components at varying lengths.

Although $\sigma(V)$ is a good indicator for small documents, it may lose its meaning for large documents. It is more reasonable to take windows of the document and apply standard deviation for these windows.

$$\sigma(V, w, k) = \sqrt{\frac{\Sigma_{i=k}^{k+w-1}(|v_i| - \mu(V, w, k))^2}{w}}$$

and

$$\mu(V, w, k) = \frac{\Sigma_{i=k}^{k+w-1}|v_i|}{n+1}$$

where w is the size of the window. To see the use of windows, consider a document where the sentences in the first half have length 10 and have length 100 in the second half. The initial standard deviation would not be able to detect this problem. The windowed standard deviation would identify this problem.

5. Syntactic Matching

Given two strings s and t where $s \in \Sigma^*$ and $t \in \Sigma^*$. The lengths of s and t are denoted with $|s|$ and $|t|$, respectively. Let $s = s_0 s_1 s_2 ... s_{n-1}$ and $t = t_0 t_1 t_2 ... t_{m-1}$. Depending on the number of consecutive matches, Look Ahead Heuristics (LAH) algorithm attempts to avoid the matching of s_i in t after a number of consecutive matches is found. LAH algorithm that checks i consecutive matches will be represented as $LAH(i)$ algorithm. We will now define i consecutive matches more formally.

Definition 5.1. (Consecutive i-match) Assume the matching of $s_{p..m}$ and $t_{q..n}$ and let $M(t, s, p, m, r)$ return the index of the first match of t_r in $s_{p..m}$ (-1 if not found). For any $q \leq r_1 < r_2 < ... < r_i \leq n$ and $M(t, s, p, m, r_k) \neq -1$, if $M(t, s, p, m, r_k) < M(t, s, p, m, r_{k+1})$ where $0 \leq k \leq i$, then r_1 is considered as a true match. Otherwise, r_1 is a false match.

5.1. LAH(1) Algorithm

$LAH(1)$ will start from t_0 and search t_0 in S until a match is found. It continues with t_1 and so on. In other words, if the algorithm was finding the LCS, first match would be part of LCS. LAH(1) algorithm is satisfied with a match. Consider the example given in Fig. 6. In this example, t_2 and s_3 are considered as a match in consecutive 1-match. Actually, this is an incorrect match. Although this matching also gives a difference between S and T, it is semantically incorrect.

The algorithm given in Algorithm 5.1 uses two match pointers, S_{future} and T_{future} where the first mismatch occurred since the last correct match in strings

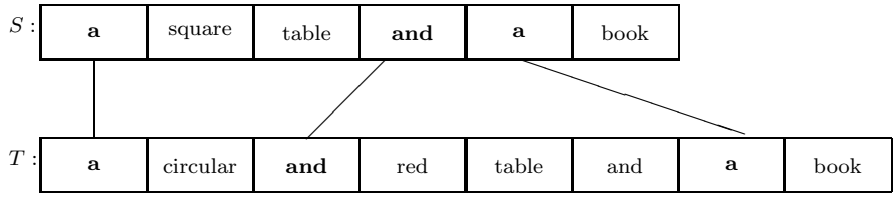


Fig. 6. LAH(1) sample matching.

S and T, respectively. The number of comparisons required for LAH(1) algorithm for the example given in Fig. 5 is 17. The number of comparisons required for dynamic programming is 48. The dynamic programming approach correctly finds the difference between two strings.

Algorithm 5.1. The implementation of the LAH(1) algorithm.

Procedure $Match\_LAH\_1(s,t)$
IN: $S = s_0 s_1 s_2 ... s_{n-1}$, $T = t_0 t_1 t_2 ... t_{m-1}$
begin
$S_{old} = S_{future} = T_{old} = T_{future} = 0$;
for $T_{future} = 1$ to $m-1$ **do**
$S_{future} = M(t, s, S_{old}, n-1, T_{old})$
if S_{future} exists **then**
$Report(t, s, S_{old}, S_{future}, T_{old}, T_{future})$
$S_{old} = S_{future}$
$T_{old} = T_{future}$
endif
endfor
end

5.2. LAH(2) Algorithm

The major drawback of LAH(1) algorithm is the assumption of the first match as the correct match. In Fig. 5, the match of t_2 and s_3 is incorrect. Instead, the match of t_5 and s_3 is a correct match. This problem can be resolved with LAH(2) algorithm. The line $t_2 - s_3$ intersects with the line t_4-s_2 (Fig. 7). The match of t_2 and s_3 was an incorrect match.

We use three counters for each string: S_{old}, $S_{current}$, and S_{future} for string S and the matching counters are T_{old}, $T_{current}$, and T_{future} for string T, respectively (i.e., $s_{old} = t_{old}$, $s_{current} = t_{current}$ and $s_{future} = t_{future}$). If the algorithm is iterating on T, it must make sure that $S_{future} > S_{current}$. If this condition is false, the match for current counters was wrong and the current counters get the values of future counters for each string.

Once the condition ($S_{future} > S_{current}$) is satisfied, we need to update the old and current pointers in both strings (e.g., $S_{old} = S_{current}$ and $S_{current} = S_{future}$) and then search for the future counters. The algorithm is given in Algorithm 5.2. The $findMatch$ procedure finds the first match starting from T_{future}. The number of comparisons that is required is 21 for the example in Fig. 7.

Since only one string is used for iteration and the other is used for comparison,

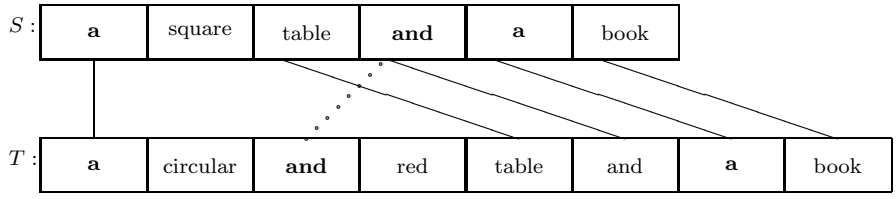


Fig. 7. LAH(2) sample matching.

every match cannot guarantee that it is a correct match. In some cases, especially when there is insertion or deletion, the match might be incorrect.

Algorithm 5.2. The implementation of the LAH(2) algorithm.

Procedure $Match\_LAH\_2(s,t)$
IN: $s = s_i s_{i+1} s_{i+2} ... s_{j-1}$ and $t = t_p t_{p+1} t_{p+2} ... t_{q-1}$
begin
$S_{old} = S_{current} = S_{future} = i - 1$
$T_{old} = T_{current} = T_{future} = j - 1$
while $(T_{future} < q)$ **do**
begin
$S_{future} = findMatch(s, t, i, j, p, q, S_{old} + 1, \&Tfuture)$
verify the match
if not verified **then**
reset S_{future}
endif
while $(S_{future} \leq S_{current}$ && $T_{future} < q)$ **do**
begin
if S_{future} exists **then**
$tempS_{old} = S_{current}$
$S_{current} = S_{future}$
$T_{current} = T_{future}$
endif
$S_{future} = findMatch(s, t, i, j, p, q, S_{old} + 1, \&Tfuture)$
verify the match
if not verified **then**
reset S_{future}
endif
increment T_{future}
endwhile
report the difference
$S_{old} = S_{current}$
$S_{current} = S_{future}$
$T_{old} = T_{current}$
$T_{current} = T_{future}$
endwhile
end

These cases need to be verified. We use a verification patch to handle these cases. We investigate and solve one major problem during verification: *incorrect*

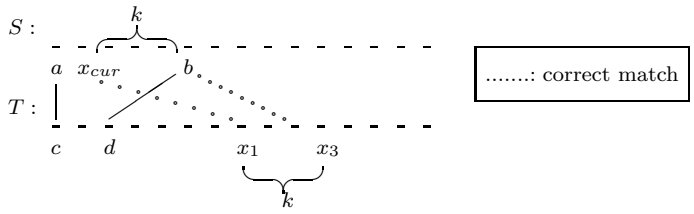


Fig. 8. Verification process: incorrect substring match.

substring match. Incorrect substring match usually occurs when a replica of a token is inserted. The distances between counters (T_{future} and $T_{current}$ or S_{future} and $S_{current}$) are more than a threshold τ_1. Let $a = S_{current}$, $b = S_{future}$, $c = T_{current}$, and $d = T_{future}$. In this case, $s_{a..b}$ is matching with $t_{c..d}$ and $d - c > \tau_1$ or $b - a > \tau_1$. This threshold does not guarantee an incorrect match but a possible indication of an incorrect match. This either indicates insertion of $t_{c..d}$, deletion of $s_{a..b}$, or substitution of $s_{a..b}$ with $t_{c..d}$. Let's consider the substitution case.

The original LAH(2) algorithm takes one string and matches the tokens of the string in the other. The problem with LAH(2) is matching with a wrong occurrence of a token. If there is a substitution, no s_k ($a \leq k \leq b$) should match t_r ($c \leq r \leq d$). The verification starts with finding the first match from $T_{current}$ of T and $X_{cur}(= S_{current}+1)$ of S and to increment X_{cur} until a match is found. Let X_1 (a possible T_{future}) be the index of the corresponding match in S. If the original match is correct, X_{cur} should be greater than S_{future}. We also search the s_{b-1} in T. If the difference between both matches is the same, this indicates the existence of incorrect substring match. To resolve this problem, S_{future} is reset. The algorithm is given in Algorithm 5.3. In the algorithm, the boundary situations are not considered where the counters may be -1 (does not exist a match) to keep the algorithm simple.

Algorithm 5.3. The implementation of the verification algorithm.

Function *Boolean* $verify(s, t, T_{current}, T_{future}, S_{current}, S_{future})$
IN: $s = s_i s_{i+1} s_{i+2} ... s_{j-1}$ and $t = t_p t_{p+1} t_{p+2} ... t_{q-1}$
begin
if $S_{future} - S_{current} > \tau_1$ or $T_{future} - T_{current} > \tau_1$ **then**
$X_{cur} = S_{current} + 1$;
$X_1 = findMatch(t, s, p, q, i, j, T_{current} + 1, \&X_{cur})$
$X_{old} = X_{cur}$;
increment X_{cur};
$X_2 = findMatch(t, s, p, q, i, j, T_{current} + 1, \&X_{cur})$
if $(X_{cur} \leq S_{future})$ **then**
$X_1 = minimum(X1, X2)$;
endif
$X_{cur} = S_{future} - 1$;
$X_3 = findMatch(t, s, p, q, i, j, T_{future} + 1, \&X_{cur})$
if $s_{Sfuture-1-X3+X1} = t_X 1$ **then**
$S_{future} = -1$;

```
        return false
      endif
    endif
    return true
  end
```

5.3. Discussion for LAH(k) Algorithm

There are two stages in the LAH algorithm: matching and verification. The matching component indicates whether current matching is likely to be correct or not. If k-consecutive match returns true, this is a very good indication that the current match is a good match. However, one of the most important parts of the LAH algorithm is the verification component. For each k, a verification algorithm needs to be developed. For this type of verification, the cases where LAH(k-1) fails have to be detected and LAH(k) verification process needs to analyze these cases so that LAH(k) will have advantage over LAH(k-1). The performance of LAH(k) depends on the power of its verification algorithm. In other words, it depends on the interpretation of the output of k-consecutive match when k-consecutive match fails.

If there is repetition of a word within next k words, k-consecutive match fails. Large k increases the probability of having duplicate words within a window of k words. Now consider, out of k matchings, one of them is out of order. How should the current match be considered: good or bad? If k is large, it should be a good match. However, as the number of out-of-order matchings increases the decision on a good or bad match becomes more difficult. Moreover, there is no straightforward strategy to handle that. In addition, we do not want to devote computing time to decide a good or a bad match in a more complex environment.

It is possible to ignore verification component to estimate the performance, and an empirical ideal k value might be determined. It would be possible to see which k outperforms the other. However, such a statistics or theoretical analysis on k value is not useful because any LAH with verification would outperform all of them.

Our approach on LAH(2) is actually comprehensive consecutive-match and verification algorithm. If the incorrect substring match is considered, our verification algorithm is checking actually a window almost any size. Comparison of our algorithm with another LAH(k) lacking verification is not a fair comparison.

5.4. Reporting

The report of differences should be given in an easily understandable way. The differences are given in terms of insertion, deletion, and substitution. Sometimes the difference can only be reported one way (e.g. just insertion). It may also be a permutation of substitution and insertion or a permutation of a substitution and deletion. In cases where a permutation is possible, we ignore deletion and insertion, and report the result as only in terms of substitution (e.g. a long string is substituted with a short string). The algorithm for reports is given in Algorithm 5.4.

Algorithm 5.4. The implementation of the reporting.

Procedure *Report*(S, T, p, q, j, k)
IN: $S = s_0 s_1 s_2 ... s_n$, $T = t_0 t_1 t_2 ... t_m$
begin
if $(q - p) = 1$ **then**
if $(k - j) > 1$ **then**
$t_{(j+1)..k}$ is inserted
endif
elseif $(k - j) > 1$ **then**
$s_{(p+1)..q}$ is substituted with $t_{(j+1)..k}$
else
$s_{(p+1)..q}$ is deleted
endif
end

Although it is rare, it is possible that structural mapping is not correct. In other words, a sentence is not mapped to the correct set of sentences. Such cases would output insertion and deletion of the same sequence. For example, s_0=*abcde* and s_1=*abcde*. Assume that structural mapping produces incorrect mapping by mapping *abc* in s_0 to *ab* in s_1; and *de* in s_0 to *cde* in s_1. In the first mapping (*abc* - *ab*), it is reported that *c* is deleted from s_0. In the second mapping (*de* - *cde*), it is reported that *c* is inserted. However, an example of correct mappings would be mapping *ab* to *ab*, and mapping *cde* to *cde*. We overcome this problem by checking whether there is a insertion/deletion at the of the string. If there is an insertion/deletion at the end of a string, it is not reported and that difference is moved to the beginning of the next mapping. In the original incorrect mapping, *c* was expected to be deleted. We do not report it right away, but we start the next matching from this point. In other words, *c* is assumed to be the head of *de*. Instead of matching, *de* to *cde*, *cde* (i.e., starts from *c*) is matched against *cde*

After the results are reported, the older document can be replaced with its difference from the new document using *delta* encoding. Whenever the older document needs to be retrieved, the older document can be generated using the *delta coding* based on the new document. It is better to maintain the latest document, since the latest document may need to be accessed more than the older document. In this way, numerous document generations using delta coding may be avoided while saving significant space.

6. Experiments and Analysis

6.1. Data sets

We have conducted our experiments on two different types of datasets: DBWorld (DBWORLD, 2007) and Wikipedia (Wikipedia, 2007).

DBWorld Data Set. We have chosen a subset of the messages that are sent by DBWorld. DBWorld messages form a good set of data with duplicate and update messages. Among 908 messages in chosen DBWorld data set, we have identified the similar messages and present our experiments on 271 comparisons. 19% of these comparisons yield edit distance of 0.

Wikipedia Data Set. There are four major reasons why we have also chosen Wikipedia:

– It is publicly available.
– It keeps history of all updates made to a document.
– It has already the wikimedia diff comparison function (Wikimedia Diff, 2007) included on their web site.
– Some documents are organized hierarchically.

We have created 9 datasets with different sizes and number of sections. The documents contain information about "answering machine", "rainbow", "computer", "George W. Bush", "hubble", "Ramazan", "JPEG", "semantics", and "Huntsville". Wikipedia allows to download at most 100 documents in XML format per search item.

Wikipedia documents are sometimes composed of sections. This organization allows us to check how *S2S* performs if sections are allowed. The availability of wikimedia diff provides an opportunity for us to compare our algorithm against a good, new, and in-use wikimedia diff algorithm. Unfortunately, we were not able to find much documentation about wikimedia diff except the comments in the code and help document.

6.2. Preprocessing

The first stage is the generation of frequency matrix for DBWorld data set. All the email messages were parsed using Lex & Yacc and created our database for our experiments. During parsing we eliminated all the extra spaces, irrelevant characters (like "==========") to build a sequence of words for DBWorld data set. We have divided the text using two punctuation marks "." and "?". We treat email addresses (e.g., raygun@cs.uah.edu) or web addresses (e.g., http://www.cs.uah.edu) as a single word by ignoring the "." in the string. However, since this division algorithm is not accurate, the "." at the end of abbreviations is also considered as an end of sentence. During frequency matrix generation we eliminated the stop words like - are, on, an, the, is, a, why, when, who and some more. For our purposes, we assumed that these words do not affect the similarity of two documents. We have not used LSI in our experiments since frequency matrix is enough to serve for our purposes. We have used the *Text* node for Wikipedia documents in XML format.

6.3. Experimental Analysis

6.3.1. String Comparisons and Granularity

Number of String Comparisons. Each word is searched for a limited number of sentences based on the output of *TraceMatch* algorithm in Section 4. This avoids the search of a new word till the end of the other document. A word may need to be matched for any word in the other document in traditional matching algorithms.

Fig. 9 displays the graph for number of string comparisons on "George W. Bush" data set. This graph shows that the number of string comparisons for our hierarchical and optimized matching algorithms is almost the same for all except at the word level. The number of string comparisons might be more than the number of comparisons in traditional string matching since a word may need

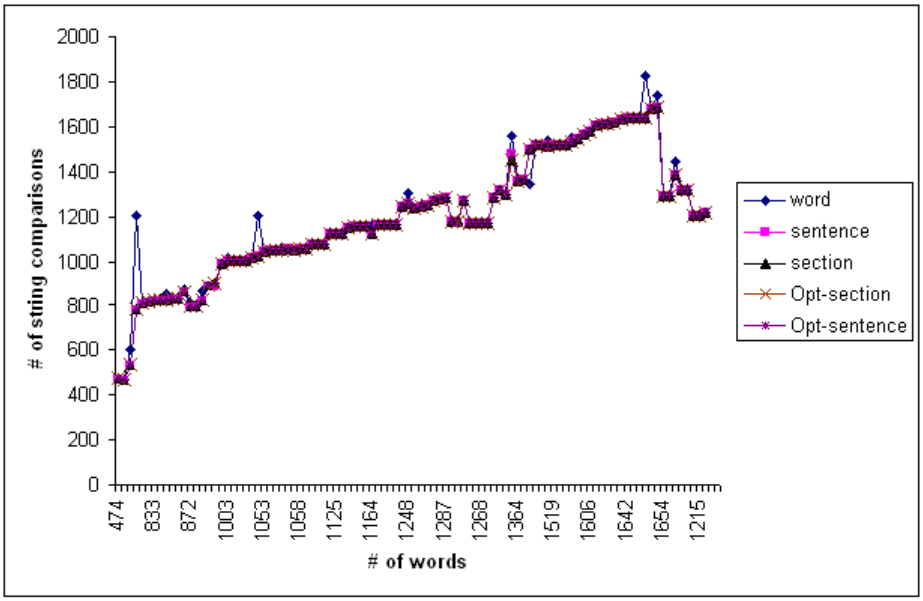


Fig. 9. The graph for the number of string comparisons for "George W. Bush" data set.

to be compared against all the words in the other document. Fig. 10 shows the ratio of the number of string comparisons to the number of words on "George W. Bush" data set. The maximum number of words represents $max(|S|, |T|)$ whereas minimum number of words represents $min(|S|, |T|)$. The average ratio for maximum number of words is 0.982 whereas it is 1.006 for minimum number of words. The average of these values is 0.994. For similar documents, it can be stated that our algorithm has linear complexity with respect to the number of words. In Fig. 10, the lower values indicate possible deletion/insertion at the of a string whereas high values indicate the number of comparisons due to verification. The advantage of our algorithm is its limiting the number of unnecessary matching by using structural mapping.

Granularity. The "answering machine" document had initially one section, and the final number of sections is eight. The section level algorithm attempts to match at the section level. However, if the section level mapping is not successful due to the same number of sentences in consecutive sections, the structural mapping is applied at the sentence level. Fig. 11 shows the number of cell computations for sentence and (optimized) section levels. For small number of sections, the section level does not have any advantage over sentence level mapping. However, the advantage of section level mapping becomes obvious at the third quarter of Fig. 11. The number of computed cells for section level is far more less than the number of computed cells for sentence level. The effect of optimization has also been shown in the figure. The number of cell computations for optimized section level is 48.4 whereas it is 150.5 for non-optimized section level. The number of cell computations for sentence level is 245.1. The average number of sentences in "answering machine" data set is 17.5. The number of cell computations for optimized section level is little bit less than three times the average number of

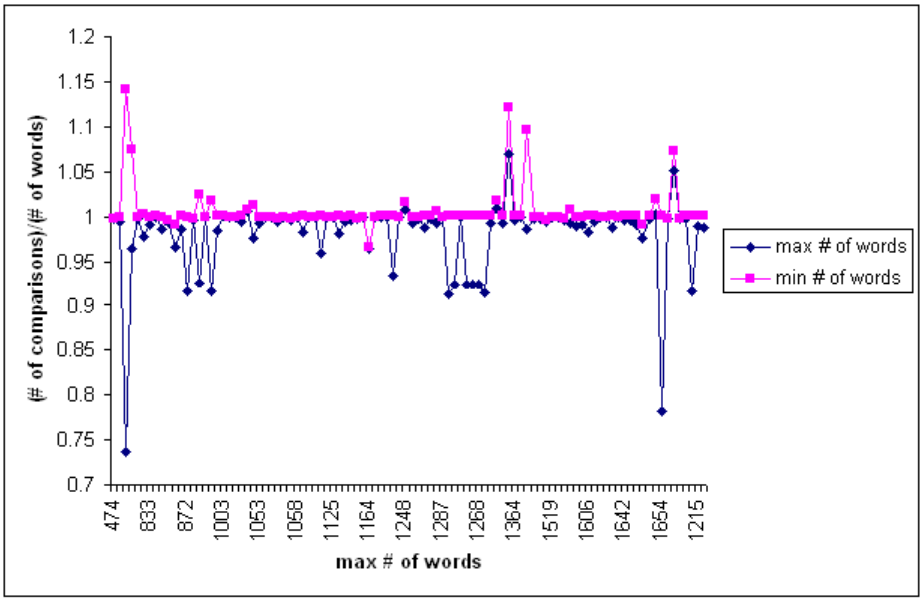


Fig. 10. The graph for the ratio of string comparisons to the number of words with respect to the number of words for "George W. Bush" data set.

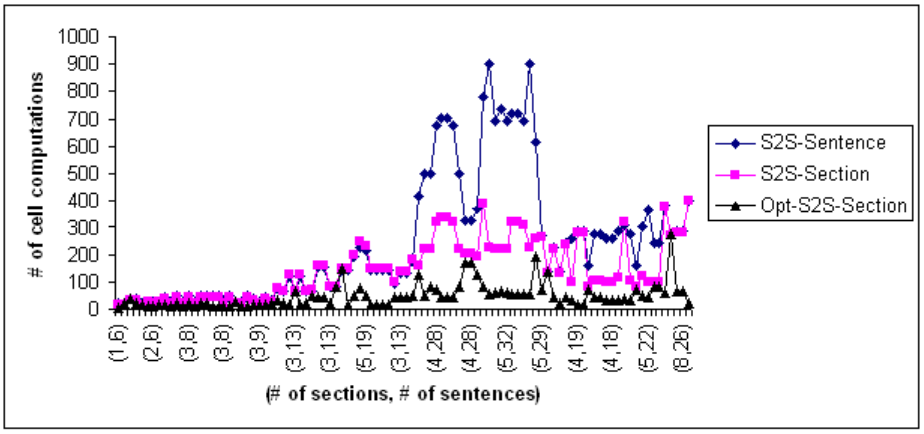


Fig. 11. The graph for the effect of granularity for "answering machine" data set.

sentences. This also shows that the number of cell computations for optimized section level is linear with the number of sentences. Note that to compute the value of a cell, we need the values for three neighboring cells. That is why the number of cell computations is very close to three times the average number of sentences.

LAH(k) without verification. We have performed experiments on the "answering machine" data set for LAH(1), LAH(2), LAH(3), LAH(4), and LAH(5).

We have got the accuracy results 85%, 89%, 77%, 38%, and 9% for LAH(1), LAH(2), LAH(3), LAH(4), and LAH(5), respectively. As we mentioned in Section 5.3, the repetitions within a window of k words degrade the performance of LAH(k) without verification. We have realized that articles (like a, an, the) are very likely to be repeated within 5 words in a document. This is the major reason for LAH(5) having low accuracy results. These results also show that LAH(2) performed better than others without any verification.

6.3.2. Comparison with Wikimedia Diff

The wikimedia diff algorithm firstly eliminates common lines from the the beginning and end of two documents. Then it uses MD5 hash function (Rivest, 1992) to index (or to check the existence of a substring in the other). Wikimedia diff starts with a sequence of string comparisons and uses hash data structure for the rest of the operations. In other words, their critical operations are performed in a different domain.

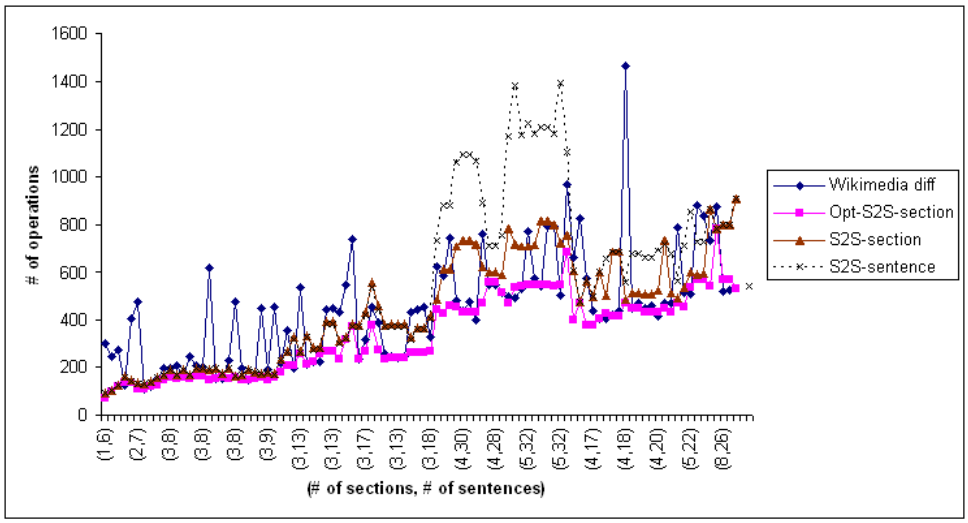
To compare *S2S* with wikimedia diff, the following steps are taken:

- Their comparison is in terms of lines while our comparison is in terms of words. When comparing two lines, the words (or characters) are actually compared. For line comparison, the number of words in a line is counted and interpreted as the number of word comparisons.
- From time to time, their algorithm checks whether a string is empty or not (because of using hash). These comparisons are also included in the total number of comparisons.
- The effect of hash function is interpreted as follows. Sometimes an input for the hash function is a complete line. Since all the characters (or words) in a line need to be processed for the key, using hash may not be more efficient than plain string comparison since hash function includes multiplication and mod functions. Whenever a hash function is called, the number of words in the input is counted.

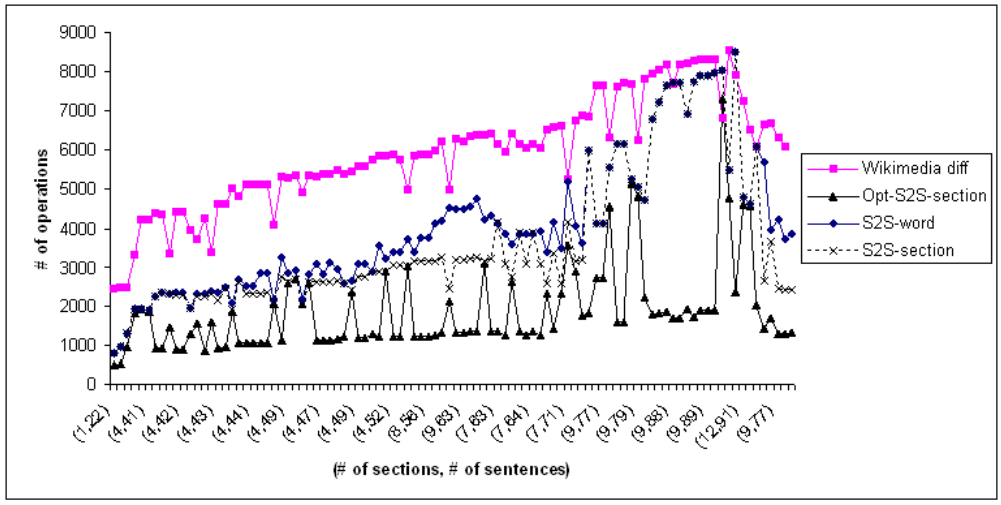
S2S has word comparisons and cell computations. On the other hand, wikimedia diff has word comparisons and number of words for MD5 hash computation. The total number of operations for *S2S* is the sum of the number of word comparisons and the number of cell computations. For wikimedia diff, the number of operations is the total number of word comparisons plus the number of words during MD5 hash computation.

Performance. Figure 12 displays number of operations on "answering machine" and "George W. Bush" data sets. In Figure 12, it can be seen that the number of operations for optimized *S2S* at section level is significantly better than that of wikimedia diff. The average number of operations for optimized *S2S* (section), non-optimized *S2S* (section), and wikimedia diff are 341, 443, and 435 for "answering machine" data set, respectively. For "George W. Bush" data set, the average number of operations for optimized *S2S* (section), non-optimized *S2S* (section), and wikimedia diff are 1865, 3267, and 5905, respectively. It should be noted that even non-optimized *S2S* is comparable with wikimedia diff.

Stability. To measure the stability of the two algorithms, we have changed the first and last word in the document. Normally, it is expected that such two changes should not affect the number of operations. Figure 13 shows the results of the stability of the algorithms on "answering machine" data set. Although



(a)



(b)

Fig. 12. The graph for the number of operations in *S2S* and wikimedia diff a) "answering machine" data set, b) "George W. Bush" data set.

S2S is almost not affected at all, there is a significant change in the number of operations for wikimedia diff. The change of average number of operations for optimized *S2S* at section level is 4.01 whereas it is 662.71 for wikimedia diff. This also shows that the performance of wikimedia diff relies on the similarity of lines at the beginning and the end of the document. However, the performance of *S2S* is significantly more stable than the performance of wikimedia diff.

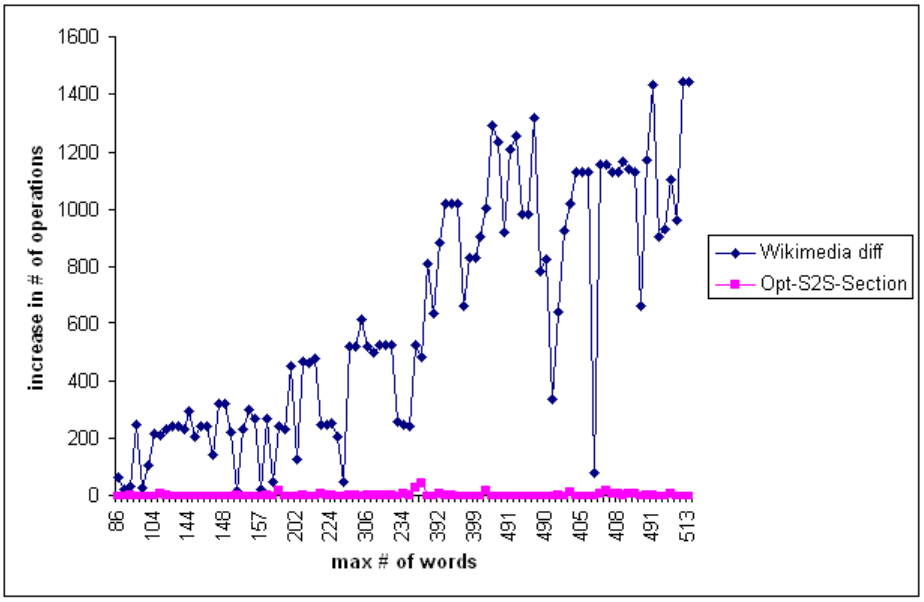


Fig. 13. The graph for the increase in the number of operations in *S2S* and wikimedia diff after the document is slightly changed.

6.3.3. Accuracy

We would like to note that it is possible to match two strings in multiple ways. Consider two strings from "answering machine" data sets: s_1 ="*An 'answering machine' is a device for automatically answering [[telephone]] calls and recording messages left by callers.*" and s_2 ="*An 'answering machine' also known as an 'answer machine' (especially in UK and British commonwealth countries), is a device for automatically answering [[telephone]] calls and recording messages left by callers.*" The update made from s_1 to s_2 is to insert "*also known as an 'answer machine', (especially in UK and British commonwealth countries),*" between *machine'* and *is a* (i.e., s_2 ="*An 'answering machine'* **also known as an 'answer machine' (especially in UK and British commonwealth countries),** *is a device for automatically answering [[telephone]] calls and recording messages left by callers.*"). However, it is also possible to match these two strings by first inserting "*also known as an 'answer*" between *answering'* and *machine'*, and then inserting "*(especially in UK and British commonwealth countries),*" between "*machine*" and "*is a*" (i.e., s_2 ="*An 'answering* **machine' also known as an 'answer** *machine'* **(especially in UK and British commonwealth countries),** *is a device for automatically answering [[telephone]] calls and recording messages left by callers.*"). In the later case, the edit distance is equal to the the the edit distance of the first case. However, the editing in the first matching is the correct editing. We accepted both of them as correct in our experiments. We should also mention that even getting a smaller editing distance does not not necessarily indicate that it is the best match between two strings.

The accuracy of *S2S* and wikimedia diff is given in Table 4. We have provided the *S2S* results for optimized section level when comparing against the wikimedia

Data set	# of docs	min words	max words	min sentences	max sentences	min section	max section	*S2S* Accuracy	wikimedia Accuracy
dbworld	271	48	1232	2	84	1	1	96.7	N/A
answering machine	100	67	413	6	33	1	8	99%	94%
computer	100	1	2764	1	122	1	16	99%	91%
george w bush	100	471	1681	22	92	1	12	99%	90%
huntsville	23	25	75	1	3	1	1	100%	91%
hubble	36	12	68	1	2	1	2	100%	100%
jpeg	100	795	1512	37	83	1	23	98%	85%
rainbow	100	1	1048	1	45	1	7	98%	94%
ramazan	17	2	417	1	26	1	1	100%	94%
semantics	100	10	380	2	22	1	7	99%	93%
Overall	676	1	2764	1	122	1	23	98.9%	92.4%

Table 4. The comparison of accuracy for wikipedia data set.

diff. It can be seen that the accuracy of *S2S* is better than the accuracy of wikimedia diff. However, it should be noted that the 1% failure of *S2S* does not indicate incorrect matching but indicates the existence of a better match between documents. *S2S* still produces valid results in those cases.

In our experiments, we have realized that two measures are helpful to determine the accuracy of the algorithm: edit ratio and difference ratio. These values are determined as follows:

$$EditRatio(s,t) = \frac{edit\ distance}{max(m,n)}$$

and

$$DifferenceRatio(s,t) = \frac{insertDist + deleteDist}{max(|m-n|,1)}$$

The edit distance checks whether the number of operations are reasonable with respect to the length of strings. The difference ratio checks whether the number of insertion or deletion operations are reasonable with respect to the difference in the length of strings. Our experiments reveal that if $EditRatio > 0.2$ or $DifferenceRatio > 2$, there is a possible mismatch of strings.

6.4. Complexity Analysis

For optimized structural mapping, we get $O(n)$ complexity. The original complexity of the structural matching algorithm is $O(mn)$ where m and n represent the number of sentences. The complexity of optimized structural matching is $O(m + n + 3 * max(m,n))$, which is equivalent to $O(n)$ when the number of sentences in two strings is close.

For syntactic matching, if two strings S and T are the same, the time complexity of our algorithm is $O(n)$ where n is the lengths of these strings. In our experiments, the length of two strings is very close to each other. The worst case for LAH algorithm occurs, when a word in T does not exist in S. In this case, the string S will be iterated to its end. If two strings S and T are totally different, the complexity of this algorithm is $O(mn)$. Since our algorithm only considers similar documents, the worst case hardly occurs.

Given two strings S and T where $S \in \Sigma^*$ and $T \in \Sigma^*$. Let $S = s_0 s_1 s_2 ... s_{n-1}$ and $T = t_0 t_1 t_2 ... t_{m-1}$. Since we compare similar messages, assume sim of the words in T exist in S where $0 \leq sim \leq 1$. If sim is 1 all the words of T exist in S. Usually, sim is more than 0.9 in our experiments. For LAH(1), the number of comparisons is $\Omega(n)$ in the best case. In this best case, S and T are exactly the same. Let g be the number of missing words of T in S and $g = (1 - sim) * m$. In the worst case, the words that do not appear in S will appear first in T. This leads to g*n comparisons for missing words. For other words, only $m - g$ comparisons are needed. So, the number of comparisons will be

$$\begin{aligned} g * (n-1) + m &= (1 - sim) * m * (n-1) + m \\ &= mn * (1 - sim) + m * sim \end{aligned}$$

If sim is guaranteed that $1 \geq sim \geq (1 - c_1/n)$, the worst case complexity will be $O(m)$ where c_1 is constant greater than or equal to 0. Since structural mapping divides the problem into smaller problems, there is an upper-bound on n. The average length of a sentence is 28 in our experiments. We get $O(m)$ complexity during our experiments. In other words, such a constant c_1 exists.

The approximate string matching algorithms like string matching with k mismatches and string matching with k errors do not provide useful information since we do not have any idea on k. Since we are interested in the differences, these algorithms will return the position where the matching starts and leave it to the user to identify the differences or will ask to use string editing distance methods to report the distance.

7. Limitations and Future Work

Our algorithm only considers three operations when matching two strings: insertion, deletion, and substitution. Our algorithm is not handling swap operations. In other words, if there is a swap operation our algorithm reports this case twice as 1) missing of words in one sentence and 2) addition in the other sentence.

For syntactic matching algorithm, we have evaluated every word whether it is a stop or a frequent word or not. In some cases, matching of frequent words causes a problem. For example, "Call for Papers" messages include program committee at the end message which includes the affiliation information. This part includes "University of" sequence for almost each member. This corresponds to a back to back match for two words. We realize that the matching of frequent words within a window should be omitted to find the correct match. Eliminating frequent words in matching solves this problem, but we leave this as a future work. The granularity of what is matched plays an important role. In this version of the algorithm, we did not consider any improvement for this part and leave it as future work.

If there are cases like swapping or frequent sequence repetition, there will be duplicate information in the report. The report can be reevaluated to eliminate these problems. At this level, the results are satisfactory and consider it as a future work.

We have encountered HTML documents in our data sets. Our algorithm usually failed for HTML documents. But we believe that if the structural mapping is performed considering the structure of HTML documents, our algorithm will also be successful.

8. Conclusion

In this paper, we have proposed our *S2S* matching approach that is composed of a heuristic structural mapping and syntactic matching algorithm. The heuristic syntactic matching algorithm is an approximation algorithm that considers i consecutive matches between two documents. The structural mapping can be improved by creating more hierarchies and performing mapping by creating a projection of documents (like table of contents) at the high levels. In cases where this algorithm fails, the complex algorithms can be used. As long as there is a way to split the documents into subdocuments, our algorithm will be successful. The structural mapping part also allows and motivates parallel processing. We are planning to conduct more experiments on XML data and genome sequences.

Acknowledgements. Our thanks to Kishore Rambatla and Nagendra Kupparavilli for the initial implementation of comparison algorithms.

References

Alfred V. Aho, Daniel S. Hirschberg, and Jeffery D. Ullman (1976). "Bounds on the Complexity of the Longest Common Subsequence Problem." J. ACM 23,1 (January 1976), 1-12.

Amir, M. Lewenstein, E. Porat (2004). Faster algorithms for string matching with k mismatches. J. Algorithms 50(2004), 257-275.

Apostolico (1996). String Editing and Longest Common Subsequences. Vol II of "Handbook of Formal Languages" (G. Rozenberg and A. Salomaa, eds.) Springer-Verlag 1996

B. Brewington and G. Cybenko (2000). Keeping up with the changing web. IEEE Computer, 33(5) May 2000

G. Chen, X. Wu, X. Zhu, A. N. Arslan, and Y. He (2006). Efficient string matching with wildcards and length constraints. Knowledge and Information Systems, Vol. 10, No. 4, November 2006, pp. 399-419.

J. Cho and H. Garcia-Molina (2000). The evolution of the web and implications for an incremental crawler. In Proc. Of 26th Int. Conf. on Very Large Data Bases, pages 117-178, September 2000.

Broder (1997). On the resemblance and containment of documents. Compression and Complexity of Sequences(SEQUENCES'97), pp. 21-29, IEEE Computer Society, 1997.

DBWORLD (2007). DBWorld, http://www.cs.wisc.edu/dbworld/ [Online; accessed 04-30-2007]

S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman (1990). Indexing by Latent Semantic Analysis. Journal of the American Society for Information Science. Vol. 41, pp. 391-407, 1990.

S. T. Dumais (1991). Improving the retrieval of information from external resources. Behavior Research Methods, Instruments and Computers. Vol. 23, pp. 229-236, 1991.

K. M. Hammouda, M. S. Kamel (2004). Document Similarity Using a Phrase Indexing Graph Model (2004). Knowledge and Information Systems, Vol. 6, No. 6, November 2004, pp. 710-727.

Daniel S. Hirschberg (1977). Algorithms for the Longest Common Subsequence Problem. J. ACM 24,4 (October 1977), 664-675.

J. Hunt, K. P. Vo, and W. Tichy (1998). Delta algorithms: An empirical analysis. ACM Transactions on Software Engineering and Methodology, 7, 1998.

Z. Li, W. K. Ng, and A. Sun (2005). Web data extraction based on structural similarity. Knowledge and Information Systems, Vol. 8, No. 4, November 2005, pp. 438-461.

D. Korn and K. -P. Vo (2002). Engineering a differencing and compression data format. In Proceedings of the Usenix Annual Technical Conference, pages 219-228, June 2002.

Webb Miller and Eugene W.Myers (1985). "A File Comparison Program." Software - Pract. Exper. 15 (1985), 1025-1040.

D.S. Milojicic, V. Kalogeraki, R. Lukose, K. Nagarajal, J. Pruyne, B. Richard, S. Rollis, Z. Xu (2002). Peer-to-Peer Computing. HP Technical Report, HPL-2002-57

Nakatsu Nakatsu, Yahiko Kambayashi and Shuzo Yajima (1982). "A Longest Common Subsequence Algorithm Suitable for Similar Text Strings." Acta Info. 18 (1982), 171-179.

G. Navarro (2001). A guided tour to approximate string matching. ACM Computing Surveys, Vol. 33, No. 1, March 2001, pp. 31-88.
R. Rivest (1992). The MD5 Message-Digest Algorithm. RFC1321, April 1992.
D. Sankoff and J. B. Kruskal (1983). Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison, Addison Wesley, Reading, MA, 1983.
A. Savant and T. Suel (2003). Server-Friendly Delta Compression for Efficient Web Access. 8th International Workshop on Web Content Caching and Distribution (WCW), September 2003.
E. Schubert, S. Schaffert, and F. Bry (2005). Structure-Preserving Difference Search for XML Documents. In Proceedings of the Extreme Markup Languages Conference 2005, August 2005, Montreal, Quebec, Canada.
A. Trigdell(2000). Efficient Algorithms for Sorting and Synchronization. PhD thesis, Australian National University, April 2000.
A. Trigdell and P. Mackerras(1996). The rsync algorithm. Technical Report TR-CS-96-05, Australian National University, June 1996
H. Wang and C. Liu (2006). Neighbourhood Counting Metric for Sequences. In Advances in Intelligent IT Active Media 2006, IOS Press 2006, pp. 243-260
Wikimedia Diff(2007), Wikimedia, Meta-Wiki, http://meta.wikimedia.org/wiki/Diff, [Online; accessed 04-30-2007]
Wikipedia (2007), The Free Encyclopedia, http://en.wikipedia.org, [Online; accessed 04-30-2007]