Web Mining: Identifying Document Structure for Web Document Clustering

by

Khaled M. Hammouda

A thesis presented to the University of Waterloo in fulfillment of the thesis requirement for the degree of Master of Applied Science in Systems Design Engineering

Waterloo, Ontario, Canada, 2002

©Khaled M. Hammouda 2002
I hereby declare that I am the sole author of this thesis.

I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Khaled M. Hammouda

I authorize the University of Waterloo to reproduce this thesis by photocopying or other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Khaled M. Hammouda
The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.
Abstract

Information is essential to us in every possible way. We rely daily on information sources to accomplish a wide array of tasks. However, the rate of growth of information sources is alarming. What seemed convenient yesterday is not convenient today. We need to sort out how to “organize” information.

This thesis is an attempt to solve the problem of organizing information, specifically organizing web information. Because the largest information source today is the World Wide Web, and since we rely on this source daily for our tasks, it is of great interest to provide a solution for information categorization in the web domain.

The thesis presents a framework for web document clustering based in major part on two very important concepts. The first one is the web document structure, which is currently ignored by many people. However, the (semi-)structure of a web document provides significant information about the content of the document. The second concept is finding the relationships between documents based on local context using a new phrase matching technique, so that documents are indexed based on phrases, rather than individual words as it is widely used now.

The combination of these two concepts creates an underlying model for robust and accurate document similarity calculation that leads to much improved results in web document clustering over traditional methods. To make the approach applicable to online clustering, an incremental clustering algorithm guided by the maximization of cluster cohesiveness is also presented. The results show significant improvement of the presented web mining system.
Acknowledgements

I am indebted to the generous help of my supervisor Professor Mohamed Kamel for his support and provision of this work. He is a source of inspiration for innovative ideas, and his kind support is well known to all his students and colleagues.

I would like also to thank Dr. Yang Wang, my thesis reader, for his input and directions for many issues involved in this work, and Professor Fakhreddine Karray, my thesis reader, for his kind and generous support.

This work has been partially funded by the NSERC strategic project grant on “Co-operative Knowledge Discovery”, led by Professor Kamel, my supervisor.

I would like also to thank all my colleagues in the PAMI research group at the University of Waterloo. They have been helpful in many situations and the knowledge we shared with each other was so valuable to the work presented in this thesis.
Contents

1 Introduction 1
  1.1 Motivation ................................................. 2
  1.2 The Challenge ............................................ 3
  1.3 Proposed Methodology ................................. 5
    1.3.1 Web Document Structure Analysis ................. 6
    1.3.2 Document Index Graph—A Document Representation Model 7
    1.3.3 Phrase-based Similarity Calculation ............... 7
    1.3.4 Incremental Document Clustering ................. 7
  1.4 Thesis Overview ....................................... 8

2 Document Clustering 9
  2.1 Properties of Clustering Algorithms ................. 10
    2.1.1 Data Model ........................................ 10
    2.1.2 Similarity Measure ................................ 13
    2.1.3 Cluster Model ..................................... 15
  2.2 Document Clustering .................................... 16
    2.2.1 Hierarchical Clustering ........................... 17
    2.2.2 Partitional Clustering .............................. 20
    2.2.3 Neural Networks and Self Organizing Maps—WEBSOM 22
    2.2.4 Decision Trees .................................... 22
    2.2.5 Statistical Analysis ............................... 23
  2.3 Cluster Evaluation Criteria ............................ 23
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1.2</td>
<td>DC-tree Clustering</td>
<td>57</td>
</tr>
<tr>
<td>5.2</td>
<td>Similarity Histogram-based Incremental Clustering</td>
<td>58</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Similarity Histogram</td>
<td>58</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Creating Coherent Clusters Incrementally</td>
<td>60</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Dealing with Insertion Order Problems</td>
<td>62</td>
</tr>
<tr>
<td>6</td>
<td>Experimental Results</td>
<td>65</td>
</tr>
<tr>
<td>6.1</td>
<td>Experimental Setup</td>
<td>66</td>
</tr>
<tr>
<td>6.2</td>
<td>Effect of Phrase-based Similarity on Clustering Quality</td>
<td>67</td>
</tr>
<tr>
<td>6.3</td>
<td>Incremental Clustering</td>
<td>70</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Evaluation of Document Re-assignment</td>
<td>71</td>
</tr>
<tr>
<td>7</td>
<td>Conclusions and Future Research</td>
<td>73</td>
</tr>
<tr>
<td>7.1</td>
<td>Conclusions</td>
<td>73</td>
</tr>
<tr>
<td>7.2</td>
<td>Future Research</td>
<td>74</td>
</tr>
<tr>
<td>A</td>
<td>Implementation</td>
<td>77</td>
</tr>
</tbody>
</table>
List of Tables

3.1 Document Information in the HEAD element .................................................. 32
3.2 Document Body Elements ................................................................................. 33
3.3 Levels of Significance of Document Parts ......................................................... 35
4.1 Frequency of Phrases ......................................................................................... 48
6.1 Data Sets Descriptions ...................................................................................... 67
6.2 Phrase-based Clustering Improvement ............................................................... 68
6.3 Proposed Clustering Method Improvement ....................................................... 70
A.1 Classes Description ............................................................................................ 79
List of Figures

1.1 Intra-Cluster and Inter-Cluster Similarity .......................... 3
1.2 Proposed System Design ............................................. 6

2.1 A sample dendogram of clustered data using Hierarchical Clustering 18

3.1 Identifying Document Structure Example .......................... 34
3.2 Document Cleaning and Generation of XML Output .............. 37

4.1 Example of the Document Index Graph ............................. 45
4.2 Incremental Construction of the Document Index Graph .......... 47

5.1 Cluster Similarity Histogram ........................................ 59

6.1 Effect of Phrase Similarity on Clustering Quality ................. 69
6.2 Quality of Clustering Comparison ................................... 72

A.1 System Architecture .................................................. 78
List of Algorithms

4.1 Document Index Graph construction and phrase matching . . . . 49
5.1 Similarity Histogram-based Incremental Document Clustering . . 62
Introduction

Information is becoming a basic need for everyone nowadays. The concept of information, and consequently communication of information, has changed significantly over the past few decades. The reason is the continuous awareness of the need to “know”, “collaborate”, and “contribute”. In every one of these tasks information is involved. We receive information, exchange information, and provide information. However, with this continuous growth of awareness and the corresponding growth of information, it has become clear that we need to organize information in such a way that will make it easier for everyone to access various types of information. By “organize” we mean to establish order among various information sources.

For the past few decades or so there has been a tremendous growth of information due to the availability of connectivity between different parties. Thanks to the Internet everyone now has access to a virtually endless sources of information through the World Wide Web (WWW or web for short). Consequently, the task of organizing this wealth of information is becoming more challenging every day. Had the different parties agreed on a structured web from the very beginning it would have been much easier for us to categorize the information properly. But the fact is that information on the web is not well structured, or
rather ill-structured. Due to this fact, many attempts have been made to categorize the information on the web (and other sources) so that easier and organized access to the information can be established.

1.1 Motivation

The growth of the world wide web has enticed many researchers to attempt to devise various methodologies for organizing such a huge information source. Scalability issues come into play as well as the quality of automatic organization and categorization. Documents on the web have a very large variety of topics, they are differently structured, and most of them are not well-structured. The nature of the sites on the web vary from very simple personal home pages to huge corporate web sites, all contributing to the vast information repository.

Search engines were introduced to help find the relevant information on the web, such as Google, Yahoo!, and Altavista. However, search engines do not organize documents automatically, they just retrieve related documents to a certain query issued by the user. While search engines are well recognized by the Information Retrieval community, they do not solve the problem of automatically organizing the documents they retrieve.

The problem of categorizing a large source of information into groups of similar topics is still unsolved. The real motivation behind the work in this thesis is to help in the resolution of this problem by taking one step further toward a satisfactory solution. The intention is to create a system that is able to categorize web documents effectively, based on a more informative representation of the document data, and targeted towards achieving high degree of clustering quality.
1.2 The Challenge

This section formalizes the problem and states the related restrictions or assumptions.

The problem at hand is how to reach a satisfying organization of a large set of documents of various topics. The problem statement can be put as follows:

*Problem Statement:* Given a very large set of web documents containing information of various topics (either related topics or mutually exclusive topics), group (cluster) the documents into a number of categories (clusters) such that: (a) the similarity between the documents in one category (*intra-cluster similarity*) is maximized, and (b) the similarity between different categories (*inter-cluster similarity*) is minimized. Consequently the quality of categorization (clustering) should be maximized.

![Figure 1.1: Intra-Cluster and Inter-Cluster Similarity](image-url)
The statement clearly suggests that given this large corpus of documents, a solution to the problem of organizing the documents has to produce a grouping of the documents such that documents in each group are closely related to each another (ideally mapped to some topic where all the documents in the group are related to that topic), while the documents from different groups should not be related to each other (i.e. of different topics). Figure 1.1 illustrates this concept.

The problem suggests that clustering of documents should be *unsupervised*; i.e. no external information is available to guide the categorization process. This is in contrast with a *classification* problem, where a “training” step is needed to build a *classifier* using a training set of labelled documents. The classifier is then used to classify unseen documents into their predicted classes. Classification is a *supervised* process. The intention of clustering systems is to group related data without any training by finding inherent structure in the data.

The problem is directly related to many research areas, including *Data Mining*, *Text Mining*, *Knowledge Discovery*, *Pattern Recognition*, *Artificial Intelligence*, and *Information Retrieval*. It has been recognized by many researchers. Some advances toward achieving satisfying results have been made. A few of these attempts can be found in [24, 48, 53, 55, 56], where different researchers from different backgrounds have gone in different directions towards solving the problem.

It has to be noted that the task of document clustering is not a well defined task. If a human is assigned to such a task, the results are unpredictable. According to an experiment done in [37], different people were assigned to the same task of clustering web documents manually. The results of the clustering varied to a large degree from one person to another. This basically tells us that the problem does not have one solution. There could be different solutions with different results, and each one would still be a valid solution to some point, or in certain situation.

The different avenues taken to tackle this problem can be grouped in two major categories. The first is the *offline clustering* approach which basically treats the job of clustering as a batch job where the number of the documents is known and the documents are available offline for clustering. The other is *online clustering*...
where clustering is done on-the-fly for documents retrieved sequentially by a search engine for example. The latter has tighter restrictions in terms of the time of the clustering process. Generally speaking, online clustering is favored for its practical use in the web domain. But sometimes offline clustering is required for reliably categorizing a large document set into different groups for later ease of browsing or access.

1.3 Proposed Methodology

The work in this thesis is geared toward achieving high quality clustering of web documents. Quality of clustering is defined here as the degree of which the resultant clusters map to the original object classes. A high quality clustering is one that correctly groups related objects in a way very similar (or identical) to the original classification of the objects. Investigation of traditional clustering methods, and specifically document clustering, shows that the problem of text categorization is a process of establishing a relationship between different documents based on some measure. Similarity measures are devised such that the degree of similarity between documents can be inferred. Traditional techniques define the similarity based on individual words in the documents [43], but it does not really capture important information such as the co-occurrence of words and word proximity in different documents.

The work presented here is aimed at establishing a phrase-based matching method between documents instead of relying on the similarity based on individual words. Using such representation and similarity information, an incremental clustering technique based on overlapped clustering model is then established. The overlapping clustering model is essential since documents, by nature, tend to relate to multiple topics at the same time.

The overall system design is illustrated in figure 1.2. Details of system implementation, along with source code of select core classes are presented in ap-
1.3.1 Web Document Structure Analysis

The clustering process starts with analyzing and identifying the web document structure, and converting ill-structured documents into well-structured documents. The process involves rigorous parsing, sentence boundary detection, word boundary detection, cleaning, stop-word removal, word stemming, separating different parts of the documents, and assigning levels of significance to the various parts of the documents. The result is well-structured XML documents that will be used for later steps in phrase matching, similarity calculation, and clustering (see Chapter 3).

---

1XML stands for eXtensible Markup Language, a markup language specified for creating structured documents according to a DTD (Document Type Definition). More information about XML could be found on the Web at http://www.w3c.org/xml.
1.3.2 Document Index Graph—A Document Representation Model

A document representation model called the "Document Index Graph" is proposed. This graph-based model captures important information about phrases in the documents as well as the level of significance of individual phrases. Matching phrases between documents becomes an easy and efficient task provided such a model (see Chapter 4). With such phrase matching information we are essentially matching local contexts between documents, which is a more robust process than relying on individual words alone. It is taken into consideration that the model should function in an incremental fashion suitable for online clustering as well as offline clustering.

1.3.3 Phrase-based Similarity Calculation

The information extracted by the proposed graph model allows us to build a more accurate similarity matrix between documents using a phrase-based similarity measure devised to exploit the extracted information effectively (see section 4.4).

1.3.4 Incremental Document Clustering

The next step is to perform incremental clustering of the documents using a special cluster representation. The representation relies on a quality criteria called the "Cluster Similarity Histogram" that is introduced to represent clusters using the similarities between documents inside the clusters. Because the clustering technique is incremental, new documents being clustered are compared to cluster histograms, and are added to clusters such that the cluster similarity histograms are improved (see Chapter 5).
1.4 Thesis Overview

The rest of this thesis is organized into six chapters. Chapter 2 presents a re-
view of document clustering and discusses some relevant work in data clustering
in general. Document (and general) data representation models are discussed,
along with similarity measures, and the requirements for document clustering
algorithms.

Chapter 3 presents the structure analysis of documents in general, and web
documents in particular. Issues related to web document structure and how the
process of identification of document structure and the conversion to a well-
defined structure are discussed.

Chapter 4 presents a novel document representation model, the “Document
Index Graph”. Document representation using the graph model, the phrase
matching technique, and similarity measurement are discussed in this chapter.

Chapter 5 discusses the incremental clustering algorithm. The cluster similar-
ity histogram representation and the clustering algorithm itself are presented.

Chapter 6 presents the experimental results of the proposed system. Quality
of clustering and performance issues are discussed.

Chapter 7 summarizes the work presented and discusses future research di-
rections.

Finally, appendix A discusses details of the system implementation with source
code listings.
CHAPTER 2

Document Clustering

This chapter presents an overview of data clustering in general, and document clustering in particular. The properties of clustering algorithms are discussed, with the various aspects they rely on.

The motivation behind clustering data is to find inherent structure in the data, and to expose this structure as a set of groups, where the data objects within each group should exhibit greater degree of similarity (known as intra-cluster similarity) while the similarity among different clusters should be minimized [25]. There are a multitude of clustering techniques in the literature, each adopting a certain strategy for detecting the grouping in the data. However, most of the reported methods have some common features [8]:

- There is no explicit supervision effect.
- Patterns are organized with respect to an optimization criterion.
- They all adopt the notion of similarity or distance.

It should be noted that some algorithms, however, make use of labelled data to evaluate their clustering results, but not in the process of clustering itself (e.g. [10, 53]). Many of the clustering algorithms were motivated by specific
problem domains. Accordingly, there is a variation on the requirements of each algorithm, including data representation, cluster model, similarity measure, and running time. Each of these requirements more or less has a significant effect on the usability of these algorithms. Moreover, it makes it difficult to compare different algorithms in different problem domains. The following section addresses some of these requirements.

This chapter is organized as follows. Section 2.1 discusses the various properties of document clustering algorithms, including data representation, similarity measures, and clustering models. Section 2.2 presents various approaches to document clustering. Section 2.3 discusses cluster evaluation criteria. The last section (2.4) summarizes the requirements of document clustering algorithms.

### 2.1 Properties of Clustering Algorithms

Before analyzing and comparing different algorithms, we first define some of their properties, and find out the relationships with their problem domains.

#### 2.1.1 Data Model

Most clustering algorithms expect the data set to be clustered in the form of a set of $m$ vectors $X = \{x_1, x_2, \ldots, x_m\}$, where the vector $x_i$, $i = 1, \ldots, m$, corresponds to a single object in the data set and is called the feature vector. How to extract the proper features to represent a feature vector is highly dependent on the problem domain. The dimensionality of the feature vector is a crucial factor on the running time of the algorithm and hence its scalability. There exist some methods to reduce the problem dimension, such as principle component analysis. Krishnapuram et al [34] were able to reduce a 500-dimensional problem to 10-dimension using this method; even though its validity was not justified. Data representation and feature extraction are two important aspects with regard to any clustering algorithm. The rest of this section focuses on data model repre-
sentation and feature extraction in general, and their use in document clustering problems in particular.

**Numerical Data Model**

A more straightforward model of data is the numerical model. Based on the problem context, a number of features are extracted, where each feature is represented as an interval of numbers. The feature vector is usually of reasonable dimensionality, yet it depends on the problem being analyzed. The feature intervals are usually normalized so that each feature has the same effect when calculating distance measures. Similarity in this case is straightforward as the distance calculation between two vectors is usually trivial [26].

**Categorical Data Model**

This model is usually found in problems related to database clustering. Usually database table attributes are of categorical nature. Usually statistical based clustering approaches are used to deal with this kind of data. The ITERATE algorithm is such an example which deals with categorical data on statistical basis [4]. The K-modes algorithm is also a good example [23].

**Mixed Data Model**

In real world problems, the features representing data objects are not always of the same type. A combination of numerical, categorical, spatial, or text data might be the case. In these domains it is important to devise an approach that captures all the information efficiently. A conversion process might be applied to convert one data type to another (e.g. discretization of continuous numerical values). Sometimes the data is kept intact, but the algorithm is modified to work on more than one data type [4].
**Document Data Model**

Most document clustering methods use the **Vector Space Model**, introduced by Salton in 1975 [43], to represent document objects. Each document is represented by a vector $d$, in the term space, $d = \{tf_1, tf_2, \ldots, tf_n\}$, where $tf_i$, $i = 1, \ldots, n$ is the term frequency in the document, or the number of occurrences of the term $t_i$ in a document. To represent every document with the same set of terms, we have to extract all the terms found in the documents and use them as our feature vector$^1$. Sometimes another method is used which combines the term frequency with the inverse document frequency (TF-IDF) [43, 1]. The document frequency $df_i$ is the number of documents in a collection of $N$ documents in which the term $t_i$ occurs. A typical inverse document frequency ($idf$) factor of this type is given by $\log(N/df_i)$. The weight of a term $t_i$ in a document is given by:

$$w_i = tf_i \times \log(N/df_i). \quad (2.1)$$

To keep the dimension of the feature vector reasonable, only a small number of $n$ terms with the highest weights in all the documents are chosen. Wong and Fu [53] showed that they could reduce the number of representative terms by choosing only the terms that have sufficient **coverage$^2$** over the document set.

Some algorithms [27][53] refrain from using term frequencies (or term weights) by adopting a binary feature vector, where each term weight is either 1 or 0, depending on whether it is present in the document or not. Wong and Fu [53] argued that the average term frequency in web documents is below 2 (based on statistical experiments), which does not indicate the actual importance of the term, thus a binary weighting scheme would be more suitable to this problem domain.

Another model for document representation is called **N-gram** [49]. The N-gram model assumes that the document is a sequence of characters. Using a sliding window of size $n$, the original character sequence is scanned to produce

---

$^1$Obviously the dimensionality of the feature vector is always very high, in the range of hundreds and sometimes thousands.

$^2$The **Coverage** of a feature is defined as the percentage of documents containing that feature.
all $n$-character sub-sequences. The N-gram approach is tolerant of minor spelling errors because of the redundancy introduced in the resulting n-grams. The model also achieves minor language independence when used with a stemming algorithm. Similarity in this approach is based on the number of shared n-grams between two documents.

Finally, a new model proposed by Zamir and Etzioni [57] is a phrase-based approach called **Suffix Tree Clustering**. The model finds common phrase suffixes between documents and builds a suffix tree where each node represents part of a phrase (a suffix node) and associated with it are the documents containing this phrase-suffix. The approach clearly captures the information of word proximity, which is thought to be valuable for finding similar documents. However, the branching factor of this tree is questionably huge, especially at the first level of the tree, where every possible suffix found in the document set branches out of the root node. The tree also suffers a great degree of redundancy of suffixes repeating all over the tree in different nodes.

Before any feature extraction takes place, the document set is normally cleaned by removing stop-words\(^3\) and then applying a stemming algorithm that converts different word forms into a similar canonical form.

### 2.1.2 Similarity Measure

A key factor in the success of any clustering algorithm is the similarity measure adopted by the algorithm. In order to group similar data objects, a proximity metric has to be used to find which objects (or clusters) are similar. There are a large number of similarity metrics reported in the literature, only the most common ones are reviewed in this section.

The calculation of the (dis)similarity between two objects is achieved through some *distance* function, sometimes also referred to a *dissimilarity* function. Given two feature vectors $\mathbf{x}$ and $\mathbf{y}$ representing two objects it is required to find the degree of (dis)similarity between them.

\(^3\)Stop-words are very common words that have no significance for capturing relevant information about a document (such as “the”, “and”, “a”, … etc).
A very common class of distance functions is known as the *family of Minkowski* distances [8], described as:

$$\|x - y\|_p = \sqrt[p]{\sum_{i=1}^{n} |x_i - y_i|^p} \quad (2.2)$$

where $x, y \in \mathbb{R}^n$. This distance function actually describes an infinite number of the distances indexed by $p$, which assumes values greater than or equal to 1. Some of the common values of $p$ and their respective distance functions are:

- $p = 1$: Hamming Distance
  $$\|x - y\|_1 = \sum_{i=1}^{n} |x_i - y_i| \quad (2.3)$$

- $p = 2$: Euclidean Distance
  $$\|x - y\|_2 = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2} \quad (2.4)$$

- $p = \infty$: Tschebyshev distance
  $$\|x - y\|_\infty = \max_{i=1,2,...,n} |x_i - y_i| \quad (2.5)$$

A more common similarity measure that is used specifically in document clustering is the *cosine correlation* measure (used by [47, 10, 53]), defined as:

$$\cos(x, y) = \frac{x \cdot y}{\|x\| \|y\|} \quad (2.6)$$

where $(\cdot)$ indicates the vector dot product and $\| \cdot \|$ indicates the length of the vector.

Another commonly used similarity measure is the *Jaccard* measure (used by [34, 27, 17]), defined as:

$$\text{sim}(x, y) = \frac{\sum_{i=1}^{n} \min(x_i, y_i)}{\sum_{i=1}^{n} \max(x_i, y_i)} \quad (2.7)$$

which in the case of binary feature vectors could be simplified to:

$$\text{sim}(x, y) = \frac{|x \cap y|}{|x \cup y|} \quad (2.8)$$
It has to be noted that the term “distance” is not to be confused with the term “similarity”. Those terms are opposite to each other in the sense of how similar the two objects are. Similarity decreases when distance increases. Another remark is that many algorithms employ the distance function (or similarity function) to calculate the similarity between two clusters, a cluster and an object, or two objects. Calculating the distance between clusters (or clusters and objects) requires a representative feature vector of that cluster (sometimes referred to as a medoid).

Some clustering algorithms make use of a similarity matrix. A similarity matrix is a $N \times N$ matrix recording the distance (or degree of similarity) between each pair of objects. Obviously the similarity matrix is a positive definite matrix so we only need to store the upper right (or lower left) portion of the matrix.

### 2.1.3 Cluster Model

Any clustering algorithm assumes a certain cluster structure. Sometimes the cluster structure is not assumed explicitly, but rather inherent in the nature of the clustering algorithm itself. For example, the $k$-means clustering algorithm assumes spherical shaped (or generally convex shaped) clusters. This is due to the way $k$-means finds cluster centers and updates object memberships. Generally speaking, if care is not taken we could end up with elongated clusters, where the resulting partition contains a few large clusters and some very small clusters. Wong and Fu [53] proposed a strategy to keep the cluster sizes in a certain range, but it could be argued that forcing a limit on cluster size is not always desirable. A dynamic model for finding clusters irrelevant of their structure is CHAMELEON (not tested in document clustering), which was proposed by Karypis et al [30].

Depending on the problem, we might wish to have disjoint clusters or overlapping clusters. In the context of document clustering it is usually desirable to have overlapping clusters because documents tend to belong to more than one topic (for example a document might contain information about car racing and car companies as well). A good example of overlapping document cluster generation is the tree-based STC system proposed by Zamir and Etzioni [57]. Another
way for generating overlapping clusters is through fuzzy clustering where objects can belong to different clusters with different degrees of membership [34].

### 2.2 Document Clustering

Clustering documents is a form of data mining that is concerned mainly with *text mining*. As far as we know, the term text mining was first proposed by Feldman and Dagan in [12]. According to a survey by Kosala and Blockeel on web mining [33], currently the term text mining has been used to describe different applications such as text categorization [20, 50, 51], text clustering [53, 56, 5, 34, 50], empirical computational linguistic tasks [18], exploratory data analysis [18], finding patterns in text databases [12, 13], finding sequential patterns in text [36, 2, 3], and association discovery [40, 50].

Document clustering can be viewed from different perspectives, according to the methods used for document representation, document processing, methods used, and applications. From the viewpoint of the information retrieval (IR) community (and to some extent Machine Learning community), traditional methods for document representation are used, with a heavy predisposition toward the vector space model. Clustering Methods used by the IR community and Machine Learning community include:

- Hierarchical Clustering [25, 10, 29],
- Partitional Clustering (*e.g.* K-means, Fuzzy C-means) [26, 47]
- Decision trees [11, 29, 40, 54],
- Statistical Analysis, Hidden Markov Models [15, 19, 29],
- Neural Networks, Self Organizing Maps [22, 52],
- Inductive Logic Programming [9, 28],
• Rule-based Systems [45, 46]

The above mentioned methods are basically at the cross roads of more than one research area, such as database (DB), information retrieval (IR), and artificial intelligence (AI) including machine learning (ML) and Natural Language Processing (NLP). The application under consideration dictates what role the method plays in the whole system. For web mining, and document clustering in particular, it could range from an Internet agent discovering new knowledge from existing information sources, to the simple role of indexing documents for an Internet search engine.

The focus here is to examine some of these methods and uncover any constraints and benefits so that we can put different methods in proper perspective. A more detailed discussion of hierarchical and partitional clustering is presented here, since they are very widely used in the literature due to their convenience and good performance.

2.2.1 Hierarchical Clustering

Hierarchical techniques produce a nested sequence of partitions, with a single all-inclusive cluster at the top and singleton clusters of individual objects at the bottom. Clusters at an intermediate level encompass all the clusters below them in the hierarchy. The result of a hierarchical clustering algorithm can be viewed as a tree, called a dendogram (Figure 2.1).

Depending on the direction of building the hierarchy, hierarchical clustering can be either Agglomerative or Divisive. The agglomerative approach is the most commonly used in hierarchical clustering.
Agglomerative Hierarchical Clustering (AHC)

This method starts with the set of objects as individual clusters; then, at each step merges the most two similar clusters. This process is repeated until a minimal number of clusters have been reached, or, if a complete hierarchy is required then the process continues until only one cluster is left. Thus, agglomerative clustering works in a greedy manner, in that the pair of document groups chosen for agglomeration is the pair that is considered best or most similar under some criterion.

The method is very simple but needs to specify how to compute the distance between two clusters. Three commonly used methods for computing this distance are:

**Single Linkage Method** The similarity between two clusters $S$ and $T$ is calculated based on the minimal distance between the elements belonging to the corresponding clusters. This method is also called “nearest neighbor” clustering method.

$$\|T - S\| = \min_{x \in T} \|x - y\|$$

**Complete Linkage Method** The similarity between two clusters $S$ and $T$ is calculated based on the maximal distance between the elements belonging to
the corresponding clusters. This method is also called “furthest neighbor” clustering method.

\[ \| T - S \| = \max_{x \in T, y \in S} \| x - y \| \]

**Average Linkage Method** The similarity between two clusters S and T is calculated based on the average distance between the elements belonging to the corresponding clusters. This method takes into account all possible pairs of distances between the objects in the clusters, and is considered more reliable and robust to outliers. This method is also known as UPGMA (Unweighted Pair-Group Method using Arithmetic averages).

\[ \| T - S \| = \frac{\sum_{x \in T} \| x - y \|}{|S| \cdot |T|} \]

It was argued by Karypis *et al* [30] that the above methods assume a static model of the inter-connectivity and closeness of the data, and they proposed a new dynamic-based model that avoids such static model. Their system, CHAMELEON, combines two clusters only if the inter-connectivity and closeness of the clusters are high enough relative to the internal inter-connectivity and closeness within the clusters.

Agglomerative techniques are usually \( \Omega(n^2) \) due to their global nature since all pairs of inter-group similarities are considered in the course of selecting an agglomeration. The *Scatter/Gather* system, proposed by Cutting *et al* [10], makes use of a group average agglomerative subroutine for finding *seed* clusters to be used by their partitional clustering algorithm. However, to avoid the quadratic running time of that subroutine, they only use it on a small sample of the documents to be clustered. Also, the group average method was recommended by Steinbach *et al* [47] over the other similarity methods due to its robustness.
Divisive Hierarchical Clustering

These methods work from top to bottom, starting with the whole data set as one cluster, and at each step split a cluster until only singleton clusters of individual objects remain. They basically differ in two things: (1) which cluster to split next, and (2) how to perform the split. Usually an exhaustive search is done to find the cluster to split such that the split results in minimal reduction based on some performance criterion. A simpler way would be to choose the largest cluster to split, the one with the least overall similarity, or use a criterion based on both size and overall similarity. Steinbach et al [47] did a study on these strategies and found that the difference between them is insignificant, so they resorted on splitting the largest remaining cluster.

Splitting a cluster requires the decision of which objects go to which sub-clusters. One method is to find the two sub-clusters using $k$-means, resulting in a hybrid technique called bisecting $k$-means [47]. Another method based on statistical approach is used by the ITERATE algorithm [4], however, it does not necessarily split the cluster into only two clusters, the cluster could be split up to many sub-clusters according to a cohesion measure of the resulting sub-partition.

2.2.2 Partitional Clustering

This class of clustering algorithms works by identifying potential clusters simultaneously, while updating the clusters iteratively guided by the minimization of some objective function. The most known class of partitional clustering algorithms are the $k$-means algorithm and its variants. $K$-means starts by randomly selecting $k$ seed cluster means; then assigns each object to its nearest cluster mean. The algorithm then iteratively recalculates the cluster means and new object memberships. The process continues up to a certain number of iterations, or when no changes are detected in the cluster means [26]. $K$-means algorithms are $O(nkt)$, where $t$ is the number of iterations, which is considered more or less a good bound. However, a major disadvantage of $k$-means is that it assumes spherical cluster structure, and cannot be applied in domains where cluster structures
are non-spherical.

A variant of $k$-means that allows overlapping clusters is known as Fuzzy C-means (FCM). Instead of having binary membership of objects to their respective clusters, FCM allows for varying degrees of object memberships [26]. Krishna-puram et al [34] proposed a modified version of FCM called ”Fuzzy C-Medoids” (FCMdd) where the means are replaced with medoids. They claim that their algorithm converges very quickly and has a worst case of $O(n^2)$ and is an order of magnitude faster than FCM.

Due to the random choice of cluster seeds, these algorithms are considered non-deterministic as opposed to hierarchical clustering approaches. The algorithm might be executed several times before a reliable result is achieved. Some methods have been employed to find ”good” initial cluster seeds. A good example is the Scatter/Gather system [10].

One approach that combines both partitional clustering with hybrid clustering is the bisecting $k$-means algorithm mentioned earlier. This algorithm is a divisive algorithm where cluster splitting involves using the $k$-means algorithm to find the two sub-clusters. Steinbach et al [47] reported that bisecting $k$-means performance was superior to $k$-means alone, and superior to UPGMA [47].

It has to be noted that an important feature of hierarchical algorithms is that most of them allow incremental updates where new objects can be assigned to the relevant cluster easily by following a tree path to the appropriate location. STC [57] and DC-tree [53] are two examples of such algorithms. On the other hand partitional algorithms often require a global update of cluster means and possibly object memberships. Incremental updates are essential for on-line applications where, for example, search query results are processed incrementally as they arrive.
2.2.3 Neural Networks and Self Organizing Maps—WEBSOM

Honkela et al [22] introduced a neural network approach for the document clustering problem called WEBSOM that is based on Self Organizing Maps (SOM), first introduced by Kohonen in 1995 [32]. The WEBSOM is an explorative full-text information retrieval method and a browsing tool [21, 31, 35]. In WEBSOM, similar documents become mapped close to each other on a two-dimensional neural network map. The self-organized document map offers a general idea of the underlying document space. The method has been used also for browsing Usenet newsgroups.

The document collection is ordered on the map in an unsupervised manner utilizing statistical information of short word contexts. Similar words are grouped into word categories to reduce the high dimensionality of the feature vector space. Documents are then mapped to word categories where they are introduced to the SOM to automatically cluster the related documents. The final clusters are visually perceived on the resulting map.

The method achieved acceptable performance especially in terms of reducing the number of dimensions of the vector space.

2.2.4 Decision Trees

Decision trees have been used widely in classification tasks [39]. The idea behind decision trees is to create a classification tree, where each node of the tree classifies a certain attribute. An object is classified by descending down the tree, comparing the object attributes to the nodes of the tree and following the node classification. A leaf corresponds to the class to which the object belongs. Quinlan [42] introduced a widely used implementation of this idea called C4.5.

For clustering purposes, however, the process is unsupervised. The process is known as Conceptual Clustering, introduced by Michalski et al in 1983 [38]. Conceptual clustering utilizes decision trees in a divisive manner, where objects are divided into sub-groups at each node according to the most discriminant attribute of the data at this node. The process is repeated until sufficient groupings
are obtained or a certain halting criteria is obtained. The method was imple-
mented and verified to be of good performance by Biswas et. al. [4].

2.2.5 Statistical Analysis

Statistical methods have been widely used as well in problems related to docu-
ment classification and clustering. The most widely used approaches are Bayes
nets and Naive Bayes. They are normally based on a probabilistic model of the
data, and mostly used for classification rather than clustering. Primary applica-
tions include key-phrase extraction from text documents [14], text classification
[9], text categorization [11], and hierarchical clustering [19, 29].

2.3 Cluster Evaluation Criteria

The results of any clustering algorithm should be evaluated using an informative
quality measure that reflects the “goodness” of the resulting clusters. The eval-
uation depends on whether we have prior knowledge about the classification of
data objects; i.e. we have labelled data, or there is no classification for the data.
If the data is not previously classified we have to use an internal quality measure
that allows us to compare different sets of clusters without reference to external
knowledge. On the other hand, if the data is labelled, we make use of this classi-
fication by comparing the resulting clusters with the original classification; such
measure is known as an external quality measure. We review two external quality
measures and one internal quality measure here.
**Entropy**

One external measure is the entropy, which provides a measure of “goodness” for un-nested clusters or for the clusters at one level of a hierarchical clustering. Entropy tells us how homogeneous a cluster is. The higher the homogeneity of a cluster, the lower the entropy is, and vice versa. The entropy of a cluster containing only one object (perfect homogeneity) is zero.

Let $P$ be a partition result of a clustering algorithm consisting of $m$ clusters. For every cluster $j$ in $P$ we compute $p_{ij}$, the probability that a member of cluster $j$ belongs to class $i$. The entropy of each cluster $j$ is calculated using the standard formula $E_j = - \sum_i p_{ij} \log(p_{ij})$, where the sum is taken over all classes. The total entropy for a set of clusters is calculated as the sum of entropies for each cluster weighted by the size of each cluster:

$$E_P = \sum_{j=1}^{m} \left( \frac{N_j}{N} \times E_j \right)$$

(2.9)

where $N_j$ is the size of cluster $j$, and $N$ is the total number of data objects.

As mentioned earlier, we would like to generate clusters of lower entropy, which is an indication of the homogeneity (or similarity) of objects in the clusters. The weighted overall entropy formula avoids favoring smaller clusters over larger clusters.

**F-measure**

The second external quality measure is the F-measure, a measure that combines the precision and recall ideas from information retrieval literature. The precision and recall of a cluster $j$ with respect to a class $i$ are defined as:

$$P = \text{Precision}(i, j) = \frac{N_{ij}}{N_i}$$

(2.10a)

$$R = \text{Recall}(i, j) = \frac{N_{ij}}{N_j}$$

(2.10b)
where $N_{ij}$: is the number of members of class $i$ in cluster $j$,
$N_j$: is the number of members of cluster $j$, and
$N_i$: is the number of members of class $i$.

The F-measure of a class $i$ is defined as:
\[
F(i) = \frac{2PR}{P+R}
\]  
(2.11)

With respect to class $i$ we consider the cluster with the highest F-measure to be the cluster $j$ that maps to class $i$, and that F-measure becomes the score for class $i$. The overall F-measure for the clustering result $P$ is the weighted average of the F-measure for each class $i$:
\[
F_P = \frac{\sum_i (|i| \times F(i))}{\sum_i |i|}
\]  
(2.12)

where $|i|$ is the number of objects in class $i$. The higher the overall F-measure, the better the clustering, due to the higher accuracy of the clusters mapping to the original classes.

**Overall Similarity**

A common internal quality measure is the **overall similarity** and is used in the absence of any external information such as class labels. Overall similarity measures cluster cohesiveness by using the weighted similarity of the internal cluster similarity:
\[
\text{OverallSimilarity}(S) = \frac{1}{|S|^2} \sum_{x \in S} \sum_{y \in S} \text{sim}(x, y)
\]  
(2.13)

where $S$ is the cluster under consideration, and $\text{sim}(x, y)$ is the similarity between the two objects $x$ and $y$. 
2.4 Requirements for Document Clustering Algorithms

In the context of the previous discussion about clustering algorithms, it is essential to identify the requirements for document clustering algorithms in particular, which will enable us to design more efficient and robust document clustering solutions geared toward that end. The following is a list of those requirements.

2.4.1 Extraction of Informative Features

The root of any clustering problem lies in the choice of the most representative set of features describing the underlying data model. The extracted features have to be informative enough to represent the actual data being analyzed. Otherwise, no matter how good the clustering algorithm is, it will be misled by non-informative features. Moreover, it is important to reduce the number of features because high dimensional feature space always has severe impact on the algorithm scalability. A comparative study done by Yang and Pedersen [55] on the effectiveness of a number of feature extraction methods for text categorization showed that the Document Frequency (DF) thresholding method produces better results than other methods and is of lowest cost in computation. Also, as mentioned in section 2.1.1, Wong and Fu [53] showed that they could reduce the number of representative terms by choosing only the terms that have sufficient coverage over the document set.

The document model is also of great importance. The most common model is based on individual terms extracted from all documents, together with term frequencies and document frequencies as explained before. The other model is a phrase-based model, such as the one proposed by Zamir and Eztioni [57], where they find shared suffix phrases in documents using a Suffix Tree data structure.
2.4.2 Overlapping Cluster Model

Any document collection, especially those in the web domain, will tend to have documents covering one or more topics. When clustering documents, it is necessary to put those documents in their relevant clusters, which means some documents might belong to more than one cluster. An overlapping cluster model allows this kind of multi-topic document clustering. A few clustering algorithms allow overlapped clustering, including fuzzy clustering [34] and suffix tree clustering (STC) [57]. In some cases it will be desirable to have disjoint clusters when each document must belong to only one cluster; in these cases one of the non-overlapping clustering algorithms can be used, or a set of disjoint clusters could be generated from fuzzy clustering after defuzzifying cluster memberships.

2.4.3 Scalability

In the web domain, a simple search query might return hundreds, and sometimes thousands, of pages. It is necessary to be able to cluster those results in a reasonable time. It has to be noted that some proposed systems only cluster the “snippets” returned by most search engines, not the whole pages (e.g. [57]). While this is an acceptable strategy for clustering search results on the fly, it is not acceptable for clustering documents where snippets do not provide enough information about the actual contents of documents. An online clustering algorithm should be able to perform the clustering in linear time if possible. An offline clustering algorithm can exceed that limit, but with the merit of being able to produce higher quality clusters.

2.4.4 Noise Tolerance

A potential problem faced by many clustering algorithms is the presence of noise and outliers in the data. A good clustering algorithm should be robust enough to handle various types of noise, and produce high quality clusters that are not affected by noise. In hierarchical clustering, for example, the nearest neighbor and furthest neighbor distance calculation methods are very sensitive to outliers,
thus should not be used if possible. The average linkage method is most appro-
priate for noisy data.

### 2.4.5 Incrementality

A very desirable feature in a dynamic domain such as the web is to be able to up-
date the clusters incrementally. New documents should be added to their respec-
tive clusters as they arrive without the need to re-cluster the whole document set. 
Modified documents should be re-processed and moved to their respective clus-
ters, if applicable. It is noteworthy that if incrementality is achieved efficiently, 
scalability is enhanced as well.

### 2.4.6 Presentation

A clustering algorithm is as good as its ability to present a concise and accurate 
description of the clusters it produces to the user. The cluster summaries should 
be representative enough of their respective content, so that the users can deter-
mine at a glance which cluster they are interested in.
Due to the wide use of web markup language HTML, a lot of useful information resides on the web in HTML format. Even other, not so widely used, document formats are often being converted to HTML to make them easily accessible over the web. As a result, we have a very large base of knowledge available over the web.

This makes it tempting to apply data mining techniques to web documents. This area of research has been known as “Web Mining”; i.e. data mining on the web. Web Mining comes in different directions. One direction is “Web Usage Mining”, which tries to find usage patterns of data on the web, such as tracking user visiting sequence of web sites, and extracting statistical information about web site usage. Another direction is “User Profiling”, which has to do with profiling users to find their interests and build a profile that could be used by marketing companies, or even for personalizing web content for different users. The direction we are dealing with here is “Web Content Mining”, which is a form of text data mining applied to the web domain, and has to do with finding structures in the content of documents, classifying documents, and clustering documents.

Since web documents are not generally well-structured, an effort has to be
done to extract useful information from web documents to make it suitable for analysis. In this chapter I present an analysis method of web documents, their structure, and how to restructure them into a useful and usable form.

3.1 Document Structure

Web Documents in HTML (Hyper-Text Markup Language) are known to be semi-structured. The word “markup” means tagging document elements in a way that facilitates the design of document layout and providing a means of referencing from one document part to another, or even from one document to another. This essentially means that HTML is not a way of structuring the content of a document; i.e. the different document elements are not planned according to a certain schema. Hence they are “semi-structured”. Since HTML specifies the layout of the document, it is used to present the document to the user in a friendly manner, rather than specify the structure of the data in the document.

Before putting the semi-structured documents into a usable structured form, we first must analyze the structure of web documents and find out what type of information it provides us with, and how we can make use of this information to make a conversion into a usable structured form.

3.1.1 HTML Document Structure

According to the W3C specification, HTML is a non-proprietary format based on SGML (Standard Generalized Markup Language). SGML is a language for specifying document structure according to a certain schema. HTML is an application of SGML for describing web documents that are viewable by web browsers over the web. HTML uses tags to describe different parts of a document for the purpose of presentation to a user by means of a browser.

The global structure of an HTML document is composed of three parts:

1. a line containing HTML version information,
2. a declarative header section,

3. a body which contains the actual document content.

Delimiters (tags) are used to delimit each part of the document. An example of a simple document is presented here:

```html
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
  <title>My first HTML document</title>
</head>
<body>
  <p>Hello world!</p>
</body>
</html>
```

The most important thing to note about that example is that information in an HTML document is separated into “meta” information that is described inside the head element, while document “content” is put inside the body element.

**Document Head**

Meta information is of very great value to us if it contains information directly related to the contents of the document, which is the case with most tags found in the head element. The most important information that describes the content of a document are listed in table 3.1.

Although the elements described here are not rendered by a web browser as part of the document, they provide very useful information about the topic of the document in a concise manner, which is exactly what we are looking for. However, since the only mandatory element is the title element, it is not always
guaranteed that whoever created the document will provide the rest of the meta information described above.

The TITLE element is of very special interest to us, since it is used to identify the content of the document. However, not all documents have context-rich titles; many document titles do not provide much contextual background, e.g. “Introduction” instead of “Introduction to Relativity Theory”. Thus no assumption should be made about the goodness of a document title when automating web mining tasks.

**Document Body**

Document content is contained in the BODY segment. Since many HTML tags are used for document body presentation, we will focus only on the tags that provide hints on document structure rather than providing cosmetic layout.

Generally speaking, there are two types of elements that are used in the document body: (a) “inline” elements (also known as text-level), and (b) “block-level” elements. Block level elements may contain inline elements and other block-level elements, thus creating a hierarchical structure. Inline elements contain only inline elements. Inherent in this structural distinction is the idea that block elements create “larger” structures than inline elements.

Table 3.2 lists some of the body elements that are relevant to document content structure.

Some elements could be used at both the block and inline-level. These are labelled as “Mixed”. Some elements have direct relationship with document structure (such as the section heading elements h1, h2, etc.), while others are related

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>TITLE</td>
<td>Document title</td>
<td>Mandatory</td>
</tr>
<tr>
<td>META-KEYWORDS</td>
<td>Document keywords used for indexing</td>
<td>Optional</td>
</tr>
<tr>
<td>META-DESCRIPTION</td>
<td>Brief description of the document contents</td>
<td>Optional</td>
</tr>
<tr>
<td>META-AUTHOR</td>
<td>The document author</td>
<td>Optional</td>
</tr>
</tbody>
</table>

Table 3.1: Document Information in the HEAD element
Adding Structure

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIV</td>
<td>Block Content</td>
<td>Block</td>
</tr>
<tr>
<td>SPAN</td>
<td>Inline Content</td>
<td>Inline</td>
</tr>
</tbody>
</table>

Specifying Structure

| H1,...,H6 | Multi-level Section Headings | Block |
| P         | Paragraph                  | Block |
| BR        | Line Breaks               | Inline|
| EM,STRONG,DFN,CITE,ABB | Structured Text          | Inline|
| BLOCKQUOTE,Q   | Quotations                | Mixed |

Special Structures

| UL,OL,LI  | Lists                   | Block |
| DL,DT,DD  | Definition Lists        | Block |
| TABLE,THEAD,TFOOT,TBODY | Tables         | Block |
| A         | Anchors                 | Inline|
| FORM      | User-input Forms        | Block |
| IMG       | Images                  | Mixed |

Table 3.2: Document Body Elements

to document layout.

These body elements will help us in identifying which parts of the document are more significant than others, and eventually will enable us to generate a more structured document out of the original one. Figure 3.1 illustrates some of the document parts we are interested in identifying.

The semi-structure that web documents have is a result of the HTML specification. The intent was to formalize a language that serve document structure as well as document layout. The result is that web documents usually do not conform to certain structuring of the data they contain. Some people will rely on layout tags to form the structure they want instead of thinking out the proper structure. This has resulted in a proliferation of misuse of the HTML language. Thus, the rationale for restructuring web documents is that the current state of semi-structured web documents is hard to deal with directly, and needs to be mapped to a data-driven structured form first.
3.2 Restructuring Web Documents

Our objective here is to convert semi-structured documents into well-structured form that is usable for data mining. Traditional methods of text mining suggest that we clean up the document by stripping all the tags out along with any non-text data to obtain a plain text document for text mining. However, for the purpose of our work, we cannot do that. Valuable information lies in the semi-structure of web documents as described earlier. It is less plausible to treat the title of the document, for example, and the body text equally. Different document parts have different levels of significance according to different factors, including their roles as specified by certain tags, their relationship to each other, and their layout and appearance.

The process of restructuring is actually not a modification to the original document structure, but more of a “clearing things out”. The proposed idea is to assign “levels of significance” to different parts of the document according to their importance. We begin by parsing the document, extracting different parts, such as the title, keywords, description, section headings, text blocks, lists, tables,
We then proceed by assigning different levels of significance to these parts. Finally we construct a XML document having the same content as the original document, but in a well structured form, with information about the significance of each part in the document. The resultant document is made available for our web mining engine described later in Chapter 4.

3.2.1 Levels of Significance

A document segment is assigned one of the four levels of significance: (1) Very High, (2) High, (3) Medium, or (4) Low, according to its role, relationship with other parts, layout, or any combination of these factors.

<table>
<thead>
<tr>
<th>Part</th>
<th>Significance Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td>Very High</td>
</tr>
<tr>
<td>Keywords</td>
<td>Very High</td>
</tr>
<tr>
<td>Description</td>
<td>Very High</td>
</tr>
<tr>
<td>Section Headings</td>
<td>High</td>
</tr>
<tr>
<td>Captions</td>
<td>Medium</td>
</tr>
<tr>
<td>Introductory Paragraph (Abstract)</td>
<td>Medium</td>
</tr>
<tr>
<td>Significant Text (Bold, Emphasized, Italics, Colored)</td>
<td>Medium</td>
</tr>
<tr>
<td>Image Alternate Text</td>
<td>Medium</td>
</tr>
<tr>
<td>Hyper-linked Text</td>
<td>Medium</td>
</tr>
<tr>
<td>Body Text</td>
<td>Low</td>
</tr>
</tbody>
</table>

Table 3.3: Levels of Significance of Document Parts

Table 3.3 lists the various recognized document parts and their respective assigned levels of significance. This weighting scheme lays a part of the foundation for similarity scoring later on (see section 4.4). When matching phrases between different documents, we have available the levels of significance of each phrase in each document. Judging how similar documents are exploits this information by scoring higher similarities for a phrase match of highly significant phrases, while a lower score will be given for low significance phrases. For example, if we have a phrase matched in the document titles, the similarity of the documents will be higher than if the match was in the body text.
3.2.2 Structured XML Documents

XML (eXtensible Markup Language) is a standard language used for describing structured documents. It is a simplified version of the SGML language. This standard was chosen for the following reasons:

- High flexibility in designing document structure
- Document schema can be enforced if desired
- Widely used in the web domain
- Standardization

We use a very simple schema to describe the structured XML document. The following is a typical skeleton of a generated structured document:

```xml
<?xml version="1.0" ?>
<document>
  <title>...</title>
  <keywords>...</keywords>
  <description>...</description>
  <abstract>...</abstract>
  <section>
    <heading>...</heading>
    <body>
      ....
    </body>
  </section>
  ....
  <significant>...</significant>
</document>
```
3.3 Cleaning Web Documents

An important part of any text mining process is the cleaning of a document. Cleaning a document is to get rid of unwanted elements of the document. The procedure for web document cleaning involves several tasks including parsing, decoding encoded characters, removing tags, detecting word and sentence boundaries, removal of stop-words, and stemming of words. Each one is described below. The overall process is illustrated in Figure 3.2.

Figure 3.2: Document Cleaning and Generation of XML Output
3.3.1 Parsing

An HTML parser was developed for analyzing HTML markup tags. The parser is the front-end for a markup analyzer, the entity designed for the identification of document structure. The markup analyzer contains the core logic for identifying different document parts and assigning levels of significance for them according to the scheme described earlier.

The parser also takes care of decoding encoded character entities, removing white-spaces, detecting block-level structures, inline structures, and stripping tags.

3.3.2 Sentence and Word Boundary Detection

A sentence boundary detector is a module developed to locate sentence boundaries in the documents. This module is based on a paper describing text boundary analysis in Java [16]. The algorithm is based on a Finite State Machine (FSM) lexical analyzer with heuristic rules for finding the boundaries. A similar approach is used to detect word boundaries.

The input is tokenized and tokens are introduced to the FSM. Sentences acceptance states are defined according to the following regular expression rules:

\[
.*/<\text{term}>*[<\text{period}>*[<\text{end}>]*<\text{space}>*;
.*/<\text{period}>*[<\text{space}>*<\text{start}>*<\text{sent-start}>;
\]

where \(<\text{term}>\) is a set of all sentence terminator characters (e.g. “?” and “!”), \(<\text{start}>\) is starting punctuation (e.g. opening parentheses and quotes), \(<\text{end}>\) is ending punctuation (e.g. closing parentheses and quotes), \(<\text{period}>\) and \(<\text{space}>\) are self-explanatory, and \(<\text{sent-start}>\) is anything that is not in the other categories and also is not a lower case letter or digit (i.e. any character that can begin a sentence). The slash character in the second rule indicates the position of the sentence break. This implies a look-ahead strategy to allow for such mid-rule acceptance state.

Word boundary detection relies on the following regular expressions:

\[
<\text{word}>=(<\text{let}>*<\text{mid-word}>*<\text{let}>*); 
<\text{number}>=(<\text{dgt}>*<\text{mid-num}>*<\text{dgt}>*); 
<\{\text{word}\}>(<\text{number}>*<\text{word}>*<\text{number}>{}*<\text{post-num}>{}); 
<\text{pre-num}>(<\text{number}>*<\text{word}>*<\text{number}>{}*<\text{post-num}>{}); 
\]

Web Documents Structure Analysis
The first two expressions are sub-expressions used in the last two rules, to avoid complication. The last two rules are the actual regular expressions used in word-boundary detection.

Word and sentence boundary detection is necessary for the phrase matching algorithm that is described in section 4.3, since it relies on the information on original document sentences.

About 98% of the actual boundaries are correctly detected. To achieve 100% accuracy, however, requires natural language processing techniques and underlying knowledge of the data set domain, which is beyond the scope of this thesis.

### 3.3.3 Stop-word Removal

Stop-words are very common words that do not provide any useful information to us, such as “the”, “which”, “and”, etc. It is often useful to get rid of these words otherwise they might mislead the clustering process by including frequent terms that are not informative to us.

When matching phrases sometimes it is desirable to keep the stop words in the phrases, because they enhance the local context matching process by allowing longer phrases to be matched. However, care has to be taken not to match a whole-stop-words phrase because it is not useful.

Stop words are actually kept in the output XML documents. Stop word removal is done in a “lazy” manner upon matching phrases. The Document Index Graph representation described in Chapter 4 includes stop words, but when matching phrases we decide whether to match stop-words or not. It was designed this way to help in matching phrases with stop words, while being able to skip the stop words when matching single words.

### 3.3.4 Word Stemming

Word stemming is the process of converting different forms of a word into one canonical form. Words like “computer”, “computing”, “compute” are all converted to a single word “compute”. This is essential to avoid treating different variations of a word distinctly. Word stemming was done using the popular Porter stemming algorithm [41].
The task of clustering in general is most affected by how well the data to be clustered is represented. Generally speaking, a set of data will have an inherent structure in some way, and our task here is to uncover that structure in the most accurate way. Clustering documents in particular depends on how well the contents of the documents are represented, and how this representation is exploited to achieve high quality clustering. Traditional techniques focused on representing documents using the Vector Space Model (VSM) [43]. Vector-based representation techniques lack the ability to represent any relationship between the different features they represent; i.e. every feature (word) is considered independent of one another, since it breaks down sentences to their individual components (see section 2.1.1). This assumption, however, can have a negative effect on determining accurate similarity between documents. Some documents, for example, will share a certain number of words but do not necessarily belong to the same category, while other documents will share only few words and belong to the same category.

This is where phrases come in play. Phrases convey more information than a list of separate words. This is due to the fact that phrases hold context information, which is essential in determining proper similarity between documents. Although phrases do not provide us with the global context of a document (without the use of natural language processing techniques) it is still desirable to have local context information that will help
in determining the degree of similarity between any two documents. This local context information is implied by word proximity, where, for example, matching two words or more appearing next to each other provides more information than just matching each one of them separately.

This chapter describes a novel model for representing documents based on sentences, as well as individual words. The model is called the Document Index Graph. This model was designed with the ability to index sentences in documents in mind, as well as the ability to find any-length matching phrases (i.e. parts of sentences) between documents. The model also represents information about the level of significance of each sentence that is determined by the document structure analysis (see Chapter 3). With this combined information (document-sentence index and sentence level of significance) we are able to make much more accurate judgement on the similarity between documents.

In this chapter the Document Index Graph model is described as a new way of document representation for the purpose of measuring the similarity between documents, and ultimately for the purpose of document clustering. The clustering process itself is independent of the model and will be presented later in Chapter 5.

### 4.1 Document Index Graph Structure

The Document Index Graph (DIG for short) is a directed graph (digraph) $G = (V, E)$

where $V$: is a set of nodes $\{v_1, v_2, \ldots, v_n\}$, where each node $v$ represents a unique word in the entire document set; and

$E$: is a set of edges $\{e_1, e_2, \ldots, e_m\}$, such that each edge $e$ is an ordered pair of nodes $(v_i, v_j)$. Edge $(v_i, v_j)$ is from $v_i$ to $v_j$, and $v_j$ is adjacent to $v_i$. There will be an edge from $v_i$ to $v_j$ if, and only if, the word $v_j$ appears successive to the word $v_i$ in any document.

The above definition of the graph suggests that the number of nodes in the graph is the number of words in the document set, i.e. the vocabulary of the document set, since each node represents a word in the whole document set.
4.1.1 Representing Sentence Structure

Sentences are represented in the graph by linking successive nodes that represent the sentence, thus creating a path in the graph for that particular sentence. A path representing a sentence is a sequence of nodes \( \{v_1, v_2, \ldots, v_k\} \) corresponding to the words in the sentence, where \( k \) is the number of words in the sentence. Path information is stored in the nodes in the path to uniquely identify each sentence. Different sentences sharing one or more sub-phrases will have an overlapping path in common representing the shared phrases.

Nodes in the graph carry information about the documents where they appeared in, along with the sentence path information. Sentence structure is maintained by recording the edge along which each sentence continues. This essentially creates an inverted list of the documents, but with sentence information recorded in the inverted list.

The structure maintained in each node is a table of documents. Each document entry in the document table records the term frequency of the word in that document. Since words can appear in different parts of a document with different level of significance, the recorded term frequency is actually broken into those levels of significance, with a frequency count per level per document entry. This structure helps in achieving a more accurate similarity measure based on level of significance later on.

Since the graph is directed, each node maintains a list of an outgoing edges per document entry. This list of edges tells us which sentence continues along which edge. The task of creating a sentence path in the graph is thus reduced to recording the necessary information in this edge table to reflect the structure of the sentences.

The document table structure represented in each node consists of the following items:

- Document ID
- Term frequency (for different levels of significance)
- Edge table

The edge table is a list of an outgoing subset of edges \( E_d \subseteq E_v \) where \( E_v \) is the list of outgoing edges that belong to node \( v \) (which in turn is a subset of the whole edge set \( E \)). \( E_d \) is the set of outgoing edges for a specific document entry in the document table,
where this list holds the path information required to follow a certain sentence in the graph.

This representation is intended to be used for effective phrase matching. Formally, a matching phrase have the following properties:

- A matching phrase is a shared sub-sentence between two or more documents.
- A matching phrase consists of one or more consecutive words, not crossing sentence boundaries.
- A matching phrase could have different frequency and significance levels in different documents.
- A matching phrase consists of the stemmed form of the constituent words.
- A matching phrase is allowed to contain stop-words, as long as there is at least one non stop-word in the phrase.

4.1.2 Example

To better illustrate the graph structure, Figure 4.1 presents an example graph that represents three documents. Each document contains a number of sentences with some overlap between the documents. As seen from the graph, an edge is created between two nodes only if the words represented by the two nodes appear successive in any document. Thus, sentences map into paths in the graph. Dotted lines represent sentences from document 1, dash-dotted lines represent sentences from document 2, and dashed lines represent sentences from document 3. As mentioned earlier, matching phrases between documents becomes a task of finding shared paths in the graph between different documents.

It has to be noted that the graph is built incrementally. When a new document arrives, it is broken down into sentences and every sentence is processed against the graph to determine matching phrases between the current document and all the previous documents (see section 4.3 for details on the phrase matching algorithm.)

According to the given example we can produce a list of matching phrases between every document and all the previously seen documents:

- Document 1
Because matching phrases between documents is symmetrical (i.e. matching phrases between document \(i\) and document \(j\) are the same between document \(j\) and document \(i\)), we need only match the phrases between every new document and the previously seen documents to get a full matching of every document to every other document. This fits nicely in an incremental system that processes documents as they arrive from a particular source without having to wait till the whole document set is available.
4.2 Constructing the Graph

The DIG is built incrementally by processing one document at a time. When a new document is introduced, it is scanned in sequential fashion, and the graph is updated with the new sentence information as necessary. New words are added to the graph as necessary and connected with other nodes to reflect the sentence structure. The graph building process becomes less memory demanding when no new words are introduced by a new document (or very few new words are introduced). At this point the graph becomes more stable, and the only operation needed is to update the sentence structure in the graph to accommodate for the new sentences introduced.

Along with indexing the sentence structure, the level of significance of each sentence is also recorded in the graph. This allows us to recall such information when we match sentences from other documents.

The order of introducing new documents is irrelevant, as any order will lead to the same final resulting graph.

Continuing from the example introduced in section 4.1.2, the process of constructing the graph that represents the three documents is illustrated in Figure 4.2. The emphasis here is on the incremental construction process, where new nodes are added and new edges are created incrementally upon introducing a new document. Sentence structure information is maintained by updating the document tables and edge tables inside each node. The phrase matching process is done at the same time while constructing the graph, and will be explained in detail in section 4.3, where the algorithm for incremental graph building and phrase matching is explained as well.

With the phrase information stored in the graph, we have complete knowledge about all the sentences that appear in the documents processed so far. The model not only reflects the occurrence of a sentence or phrase in some of the documents, but it is also accurate enough to tell us how many times a phrase appeared in every document, i.e. the frequency of phrases, and that is for every part of a sentence. For example, suppose we have two documents containing the following sentences (we shall use simple one-letter words for the purpose of this example):

- Document 1
Figure 4.2: Incremental Construction of the Document Index Graph
By constructing a Document Index Graph for the two documents above, we can detect the exact frequency of any-length phrase. Table 4.1 lists the phrase frequency information that can be deduced from the given example.

<table>
<thead>
<tr>
<th>phrase</th>
<th>document</th>
<th>frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;a b c d&quot;</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>&quot;b c d&quot;</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>&quot;a b&quot;</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>&quot;a b&quot;</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>&quot;x y z&quot;</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>&quot;x y&quot;</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Unlike traditional phrase matching techniques that are usually used in information retrieval literature, the Document Index Graph provides complete information about full phrase matching between every pair of documents. While traditional phrase matching methods are aimed at searching and retrieval of documents that have matching phrases to a specific query, the Document Index Graph is aimed at providing complete information about the degree of overlap between every pair of documents. The information about the matching phrases frequency along with the phrases level of significance will aid in specifying the similarity between the documents using a phrase-based similarity measure that is explained in section 4.4.
4.3 Detecting Matching Phrases

In this section we shall describe the algorithm for matching phrases between documents. As mentioned earlier, the graph model described here is built incrementally. As a new document is introduced, two tasks are performed in parallel: (a) incrementally expand the graph to accommodate the new document; and (b) matching phrases between the new document and all previously processed documents. The matching process is done along the way while scanning the new document, and the document similarity calculation is essentially performed in an incremental fashion as will be described later. Once the new document is processed, we have a complete list of similarities between this document and all the other (previously seen) documents. Algorithm 4.1 describes the process of both incremental graph building and phrase matching.

**Algorithm 4.1** Document Index Graph construction and phrase matching

1: \( D \leftarrow \text{New Document} \)
2: for each sentence \( s \) in \( D \) do
3: \( w_1 \leftarrow \text{first word in } s \)
4: if \( w_1 \) is not in \( G \) then
5: Add \( w_1 \) to \( G \)
6: end if
7: \( L \leftarrow \text{Empty List} \) \{\( L \) is a list of matching phrases\}
8: for each word \( w_i \in \{w_2, w_3, \ldots, w_k\} \) in \( s \) do
9: if \( (w_{i-1}, w_i) \) is an edge in \( G \) then
10: Extend phrase matches in \( L \) for sentences that continue along \( (w_{i-1}, w_i) \)
11: Add new phrase matches to \( L \)
12: else
13: Add edge \( (w_{i-1}, w_i) \) to \( G \)
14: Update sentence path in nodes \( w_{i-1} \) and \( w_i \)
15: end if
16: end for
17: end for
18: Output matching phrases in \( L \)
The procedure starts with a new document to process (line 1). We expect the new document to have well defined sentence boundaries; each sentence is processed individually. This is important because we do not want to match a phrase that spans two sentences (which could break the local context we are looking for.) It is also important to know the original sentence length so that it will be used in the similarity calculation (section 4.4). For each sentence (for loop at line 2) we process the words in the sentence sequentially, adding new words as new nodes to the graph, and constructing a path in the graph (by adding new edges if necessary) to represent the sentence we are processing. Matching the phrases from previous documents is done by keeping a list $L$ that holds an entry for every previous document that shares a phrase with the current document $D$. As we continue along the sentence path, we update $L$ by adding new matching phrases and their respective document identifiers, and extending phrase matches from the previous iteration (lines 10 and 11). If there are no matching phrases at some point, we just update the respective nodes of the graph to reflect the new sentence path (lines 13 and 14). After the whole document is processed $L$ will contain all the matching phrases between the current document and any previous document that shared at least one phrase with the new document. Finally we output $L$ as the list of documents with matching phrases and all the necessary information about the matching phrases (length, frequency, and level of significance).

The above algorithm is capable of matching any-length phrases between a new document $D$ and all previously seen documents in roughly $O(m)$ time, where $m$ is the number of words in document $D$. Some could argue that the step at line 10, where we extend the matching phrases as we continue along an existing path, is not actually a constant-time step, because when the graph starts building up, the number of matching phrases becomes larger, and consequently when moving along an existing path we have to match more phrases, or even lookup larger tables. However, this is actually not the case because of two reasons. The first has to do with the nature of the documents themselves. It turns out that the size of the list of matching phrases becomes roughly constant even with very large document sets, due to the fact that a certain phrase will be shared by only a small set of documents; which on average tends to be a constant number. The second reason has to do with the implementation of the graph document tables and edge tables. The implementation we adopt is a constant-time lookup operation by using hash tables instead of regular arrays, which significantly reduces the time required to lookup
the document and edge tables described earlier in this chapter (but on the cost of slightly increasing the memory requirements for this particular implementation.)

---

### 4.4 A Phrase-based Similarity Measure

Finding the similarity between two documents is clearly dependent on several factors. One of them is the degree of overlap between the two documents. According to the similarity measures described in Chapter 2 the degree of similarity between two documents is usually measured as the degree of overlap between the terms in the documents. The *cosine correlation* measure considers documents as vectors of term weights and finds the cosine of the angle between the vectors. The closer the angle, the more similar the two documents are. The *Jaccard* measure finds the intersection of the terms between the two documents. The more terms in the intersection, the more the documents are similar.

In essence, most of these measures make a very strong assumption that words in the documents are independent of each other. They also do not make any use of word proximity information. Individual words could not be a good representative of a document unless they relate to each other in some way. This is where phrases appear to be more suitable for finding the degree of overlap between documents. As we mentioned earlier, phrases convey local context information, which is essential in determining an accurate similarity between documents.

Towards this end we devised a similarity measure based on matching phrases rather than individual terms. This measure exploits the information extracted from the previous phrase matching algorithm to better judge the similarity between the documents.

The phrase similarity between two documents is calculated based on the list of matching phrases between the two documents. This similarity measure is a function of four factors:

- The number of matching phrases $P$,
- The lengths of the matching phrases $(l_i : i = 1, 2, \ldots, P)$,
- The frequencies of the matching phrases in both documents $(f_{i1}$ and $f_{i2} : i = 1, 2, \ldots, P)$, and
• The levels of significance (weight) of the matching phrases in both document 
(w₁ and w₂ : i = 1, 2, . . . , P).

Because multiple occurrences of the same phrase could happen in either document, 
the matching phrase frequency in both documents has to be taken into consideration. 
The more frequent the phrase appears in both documents, the more similar they tend to be. Similarly, the level of significance of the matching phrase in both documents should 
be taken into consideration. If a matching phrase appears at different levels of significance in a document, the representative level of significance is taken as the weighted average of the different levels of the multiple occurrences.

The phrase similarity between two documents, \( d₁ \) and \( d₂ \), is calculated using the 
following empirical equation:

\[
\text{sim}_p(d₁, d₂) = \sqrt{\frac{\sum_{i=1}^{P} [g(l_i) \cdot (f_iw₁ + f_iw₂)]^2}{\sum_j |s_j₁| \cdot w₁ + \sum_k |s_k₂| \cdot w₂}}
\]  

(4.1)

where \( g(l_i) \) is a function that scores the matching phrase length, giving higher score as 
the matching phrase length approaches the length of the original sentence; \( |s_j₁| \) and \( |s_k₂| \) 
are the original sentences lengths from document \( d₁ \) and \( d₂ \), respectively. The equation rewards longer phrase matches with higher level of significance, and with higher frequency in both documents. The function \( g(l_i) \) in the implemented system was used as:

\[
g(l_i) = (|ms_i|/|s_i|)^\gamma
\]

where \( |ms_i| \) is the matching phrase length, and \( \gamma \) is a sentence fragmentation factor 
with values greater than or equal to 1. If \( \gamma \) is 1, two halves of a sentence could be matched independently and would be treated as a whole sentence match. However, by increasing \( \gamma \) we can avoid this situation, and score whole sentence matches higher than fractions of sentences. A value of 1.2 for \( \gamma \) was found to produce best results.

The normalization by the length of the two documents is necessary to be able to 
compare the similarities from other documents.
4.4.1 Combining single-term and phrase similarities

Our experiments show that the number of matching phrases between related documents is usually small, and between non-related documents is almost negligible. However, even a few number of significant matching phrases is enough to judge the two documents as being similar. This is because phrases have inherent context information, where term proximity plays an important role in indicating that documents having matching phrases share the same context to a certain degree. The more matching phrases there are, the more context matching we have. That is why the phrase similarity measure presented earlier is geared toward rewarding higher number of longer and significant phrase matches, which reflects a significant context overlap between the two documents.

However, if the similarity between documents is based solely on matching phrases, and not single-terms at the same time, related documents could be judged as non-similar if they do not share enough phrases (a typical case that could happen in many situations.) To alleviate this problem, and to produce high quality clusters, we combined single-term similarity measure with our phrase-based similarity measure. We used the cosine correlation similarity measure [43][44], with TF-IDF term weights, as the single-term similarity measure. The cosine measure was chosen due to its wide use in the document clustering literature, and since it is described as being able to capture human categorization behavior well [48].

Recall that the cosine measure calculates the cosine of the angle between the two document vectors. Accordingly our term-based similarity measure \( \text{sim}_t \) is given as:

\[
\text{sim}_t(d_1, d_2) = \cos(d_1, d_2) = \frac{d_1 \cdot d_2}{\|d_1\| \|d_2\|}
\]

where the vectors \( d_1 \) and \( d_2 \) are represented as term weights calculated using TF-IDF weighting scheme.

The combination of the term-based and the phrase-based similarity measures is a weighted average of the two quantities from equations 4.1 and 4.2, and is given by equation 4.3.

\[
\text{sim}(d_1, d_2) = \alpha \cdot \text{sim}_p(d_1, d_2) + (1 - \alpha) \cdot \text{sim}_t(d_1, d_2)
\]

where \( \alpha \) is a value in the interval \([0, 1]\) which determines the weight of the phrase simi-
larity measure, or, as we call it, the Similarity Blend Factor. According to the experimental results discussed in Chapter 6 we found that a value between 0.6 and 0.8 for $\alpha$ results in the maximum improvement in the clustering quality.

It has to be noted that the phrase-based similarity tends to have lower values compared to their term-based counterparts. This is understandable because of the nature of documents where matching phrases are going to be always fewer in number than matching terms. However, the similarity values produced by matching phrases are more accurate than those based on individual words alone. The similarity based on phrases has more discriminating power than the similarity based on individual words. By combining the two together, we achieve a very robust similarity calculation that is insensitive to noise and accurate enough to help in separating the clusters well.

Finally, the similarities calculated between a new document and the previous documents is used to construct an incremental similarity matrix that is used as input for the clustering method described in Chapter 5.
Chapter 5

Incremental Document Clustering

Clustering of web documents, as mentioned in Chapter 2, can be looked at from two perspectives, either offline clustering or online clustering. Offline clustering is usually performed on very large document sets, where clustering results are not reported in real-time, but rather made available later for accessing the identified document clusters. Online clustering, on the other hand, is concerned with clustering documents “on-the-fly”, where clustering results are made available instantly. Obviously online clustering has more restrictions in terms of the time available for clustering, and as a result the time complexity of any online clustering algorithm should be reasonable enough for real-time applications. An applications example is clustering the results of a web search engine query. Offline clustering algorithms do not have such restrictions.

In this chapter we present a brief review of incremental clustering algorithms, and introduce the proposed incremental document clustering algorithm, based on pair-wise document similarity, employed as part of the web document clustering system.
5.1 Incremental Clustering

To achieve reasonable performance, online clustering algorithms usually employ an incremental clustering strategy. Incremental clustering works by processing data objects one at a time, incrementally assigning data objects to their respective clusters while they progress. The process is simple enough, but faces several challenges, including:

- How to determine to which cluster the next object should be assigned?
- How to deal with the problem of insertion order?
- Once an object is assigned to a cluster, should its assignment to the cluster be frozen or is it allowed to be re-assigned to other clusters later on?

To be truly incremental, a clustering algorithm has to deal with these challenges in a smart way to achieve reasonable time bound. We are going to present two incremental document clustering algorithms here and analyze how they opted to solve these problems. The first one is the Suffix Tree Clustering (STC) algorithm [56], and the second is the DC-tree algorithm [53].

5.1.1 Suffix Tree Clustering

Introduced by Zamir et al [56] in 1997, the STC algorithm is claimed to be a fast $O(n \log n)$ incremental clustering algorithm. The basic idea is to build a trie (a compact tree) of phrase suffixes found in the document set. The trie structure represents each suffix as a node in the tree, where common suffixes from different documents are represented by the same node. A node branches off to children nodes when suffixes begin to differ at some point in the original phrases. An example is provided in the reference mentioned above.

New documents are incorporated into the tree incrementally by creating new nodes for unseen suffixes, and adding the document to old suffix nodes that appear in the document.
Each node is shared by a certain number of documents, which is identified as a base cluster. Base clusters are combined if they have a 50% overlap or more, creating the set of final clusters.

It could be argued that the STC algorithm is truly incremental. The algorithm incorporates new documents incrementally into the trie structure. Identifying the base clusters is also incremental. But forming the final clusters is not truly incremental. The final step involves processing of base clusters into final clusters by finding overlaps above the 50% threshold. This is an expensive step to do after every document is introduced to the algorithm.

The algorithm deals properly with the insertion order problem, since any insertion order into the suffix tree will lead to the same result. But the cost of re-combining the base clusters every time a document is inserted is not justified.

5.1.2 DC-tree Clustering

The DC-tree incremental algorithm was introduced by Wong et al [53] in 2000. The algorithm is based on the $B^+$-tree structure. Unlike the STC algorithm, this algorithm is based on vector space representation of the documents, where each document is represented as a feature vector of term weights. Most of the algorithm operations are borrowed from the $B^+$-tree operations.

Each node in the tree is a representation of a cluster, where a cluster is represented by the combined feature vectors of its individual documents. Inserting a new document involves comparison of the document feature vector with the cluster vectors at one level of the tree, and descending to the most similar cluster. The process is repeated until a leaf node is reached, or the similarity between the document and the clusters at one level does not cross a certain threshold, in which case a new cluster node is created.

The algorithm defines several parameters and thresholds for the various operations. Parameter tuning (or training) is a problem of this algorithm. The algorithm is incremental in terms of assigning new documents to their respective clusters, but it has two drawbacks. First, once a document is assigned to a cluster it is not allowed to be reassigned later to a newly created cluster. Second, which is a consequence of the first drawback, clusters are not allowed to overlap; i.e. a document can belong to only one cluster.
5.2 Similarity Histogram-based Incremental Clustering

The clustering approach proposed here is an incremental and dynamic method of building the clusters. An overlapped cluster model is adopted (see section 2.1.3 for discussion of cluster models), where a document is allowed to be assigned to more than one cluster at the same time. The key concept for the proposed clustering method is to keep each cluster at a high degree of coherency at any time. By achieving tight coherency on clusters we satisfy the requirement of maximizing intra-cluster similarity (see section 1.2).

5.2.1 Similarity Histogram

We represent the coherency of a cluster by a new concept called similarity histogram. A similarity histogram is a concise statistical representation of the set of pair-wise document similarity distribution in the cluster. The similarity histogram specifies a number of bins corresponding to fixed similarity value intervals. Each bin contains the count of pair-wise document similarities in the corresponding interval. Whenever a new similarity value between two documents is calculated, it is used to increment the similarity count in the bin in which this similarity value lies. Thus we are essentially creating a concise representation of the set of pair-wise document similarities in the cluster.

Figure 5.1 shows three histograms: a typical similarity histogram, where the distribution is almost a normal distribution; a perfect similarity histogram, where all the similarities in the cluster are all maximum; and a loose similarity histogram, where similarities in the cluster are all minimum.
Figure 5.1: Cluster Similarity Histogram
5.2.2 Creating Coherent Clusters Incrementally

Our objective is to keep each cluster as coherent as possible, which translates to maximizing the similarities inside the clusters. Ideally the similarity histogram of each cluster should be a perfect distribution. But typically we should keep the distribution of similarities in the cluster higher in the high similarity intervals.

We can achieve this by judging the effect of adding a new document to a certain cluster. If the document is going to degrade the distribution of the similarities in the clusters very much, it should not be added, otherwise it is added. We could have enforced a much tighter restriction to add documents to a cluster if it is only going to enhance the distribution of the similarities in the cluster. However, this could create a problem if we try to add a document to a cluster that has very good distribution already, but adding the document will degrade the distribution slightly. Imagine the scenario of adding a document to a perfect cluster for example; the document might have high similarity to most of the documents in the cluster, and some low similarities to the rest of the documents in the cluster. Adding the document to the cluster will degrade the similarity distribution (because it is perfect), but we do not want the cluster to reject it just because it slightly degrades the similarity distribution.

We now turn our focus on how to judge the effect of adding a document to a cluster on the cluster similarity histogram. We must first find out how to judge the quality of a similarity histogram distribution. As mentioned earlier, A cohesive cluster will have larger count of similarities in the higher similarity intervals, and smaller count of similarities in the lower similarity intervals. We judge the cohesiveness of the cluster by calculating the ratio of the count of similarities above a certain similarity threshold $S_T$ to the total count of similarities. This ratio is essentially the percentage of the similarities above the threshold. The higher this ratio is, the more coherent the cluster is.

Let $n$ be the number of the documents in a cluster. The number of pair-wise similarities in the cluster is $m = n(n + 1)/2$. Let $S = \{s_i : i = 1, \ldots, m\}$ be the set of similarities in the cluster. The histogram of the similarities in the cluster is represented as:

$$H = \{h_i : i = 1, \ldots, B\}$$

$$h_i = \text{count}(s_k) \quad s_{li} < s_k < s_{ui}$$

where $B$: the number of histogram bins,
\(h_i\): the count of similarities in bin \(i\),
\(s_{li}\): the lower similarity bound of bin \(i\), and
\(s_{ui}\): the upper similarity bound of bin \(i\).

The histogram ratio of a cluster is the measure of cohesiveness of the cluster as described above, and is calculated as:

\[
HR(C) = \frac{\sum_{i=T}^{B} h_i}{\sum_{j=1}^{B} h_j} \tag{5.2a}
\]

\[
T = \lfloor S_T \cdot B \rfloor \tag{5.2b}
\]

where \(HR\): the histogram ratio,
\(C\): the cluster under consideration,
\(S_T\): the similarity threshold, and
\(T\): the bin number corresponding to the similarity threshold.

Basically we would like to keep the histogram ratio of each cluster high. However, since we allow documents that can degrade the histogram ratio of a cluster slightly to be added, we could have a chain effect of degrading the histogram ratio by adding documents that only degrade the ratio but not enhance it, eventually bringing it to near zero; i.e. most of the similarities will be below the threshold. To prevent this chaining effect, we must set a minimum histogram ratio \(HR_{min}\) that clusters should maintain. We also do not allow adding a document that will bring down the histogram ratio significantly (even if still above \(HR_{min}\)). This is to prevent a bad document from severely bringing down cluster quality by one single document addition. A document will be added only if it does not bring the histogram ratio of a cluster more than \(\epsilon\).

We now present the incremental clustering algorithm based on the above framework (Algorithm 5.1).

The algorithm works incrementally by receiving a new document, and for each cluster calculates the cluster histogram before and after simulating the addition of the document (lines 3-5). The old and new histogram ratios are compared and if the new ratio is greater than or equal to the old one, the document is added to the cluster. If the new ratio is less than the old one by no more than \(\epsilon\) and still above \(HR_{min}\), it is added (lines 6-8). Otherwise it is not added. If after checking all clusters the document was not assigned to any cluster, a new cluster is created and the document is added to it (lines 10-13).
Algorithm 5.1 Similarity Histogram-based Incremental Document Clustering

1: \( D \leftarrow \) New Document documents

2: for each cluster \( C \) do

3: \( HR_{\text{old}} = HR(C) \)

4: Simulate adding \( D \) to \( C \)

5: \( HR_{\text{new}} = HR(\hat{C}) \)

6: if \( (HR_{\text{new}} \geq HR_{\text{old}}) \) OR \( (HR_{\text{new}} > HR_{\text{min}}) \) AND \( (HR_{\text{old}} - HR_{\text{new}} < \varepsilon) \) then

7: Add \( D \) to \( C \)

8: end if

9: end for

10: if \( D \) was not added to any cluster then

11: Create a new cluster \( C \)

12: ADD \( D \) to \( C \)

13: end if

5.2.3 Dealing with Insertion Order Problems

So far we are able to perform incremental clustering guided by a cluster cohesiveness measure. However, every incremental clustering algorithm suffers from the insertion order problem. Different document insertion orders will lead different results. A related problem is how to re-assign documents that were added earlier to newly created clusters if those documents should belong to the new clusters. Clearly this is a side effect of insertion order as well.

Our strategy here is to implement a re-assignment strategy in the algorithm, where documents that seem to be “bad” for a cluster cohesiveness will be removed from the cluster and re-assigned to other more suitable clusters. The re-assignment strategy is not done for all the documents, or else it would be a re-clustering step. The documents that are candidates to leave a cluster are the documents that their leaving the cluster will increase the cluster similarity histogram ratio; \( i.e. \) the cluster is better off without them.

The idea here is for each cluster, we keep with each document in the cluster a record of the cluster histogram ratio if the document was not in the cluster. If this value is greater than the current histogram ratio, then the document is a candidate for leaving the cluster, because, if it leaves the cluster, the cluster histogram ratio will increase.

Upon adding a new document to any cluster, we consult all the clusters for any doc-
documents that are considered candidates for leaving those clusters. If, for a certain cluster, there is a document that should leave that cluster, we first check if it can be added to any of the other clusters. If this is the case, the document is transferred from its current cluster to the other cluster(s), thus increasing the histogram ratio of the cluster. However, if there is no cluster accepts the document, it is not removed from its original cluster, to discourage creating small singleton clusters.

This essentially creates a negotiation protocol between clusters for negotiating document transfer. If a cluster sees that a document is decreasing the cluster quality, it negotiates with other clusters to take the document if it is going to improve their quality.

This strategy alleviates the problems of insertion order by allowing the re-assignment of documents. Documents are allowed to leave one cluster and join another. It also allows clusters to overlap, which was one of our primary requirements.
Experimental Results

Motivated by the information categorization problem presented earlier in the introduction to this thesis, we presented a framework for web document clustering. The confidence in the overall system performance is yet to be justified by experimental results. In this chapter we present and discuss the experiments conducted to evaluate the whole system. The results show a significant improvement in clustering quality and performance.

The experiments conducted for evaluating the system was divided into two sets. We first tested the effectiveness of the Document Index Graph model, presented in Chapter 4, and the accompanying phrase matching algorithm for calculating the similarity between documents based on phrases versus individual words only. The second set of experiments was to evaluate the performance of the incremental document clustering algorithm, presented in Chapter 5, based on the cluster cohesiveness measure using similarity histograms.

This chapter is organized as follows. Section 6.1 gives a brief outline of the experimental setup we used for our experiments. Section 6.2 discusses the effect of phrase matching on clustering quality. Incremental clustering results are presented in section 6.3. Finally some notes about the performance of the system are discussed in the last section.
6.1 Experimental Setup

Text mining experiments rely on a large corpus of documents. Many large corpora are available for text mining research. However, our interest here is in web documents in particular, not just plain text documents. As discussed in Chapter 3, we rely on the semi-structure of web documents to infer some information about the significance of different document parts. Plain text documents do not provide such information. We have to rely on web documents in our experiments.

Not so many web document corpora are available. There are no standard web document corpus. In fact most of the research done using web documents was conducted on proprietary document collections. Zamir et al [56] relied on their own web document collection, retrieved by the meta search engine Meta-Crawler. Wong et al [53] relied on documents manually collected from the Yahoo! web directory.

When research is being done on different corpora, results are very hard to compare\(^1\). However, the only option is to manually collect web documents and classify them for testing purposes. Two data sets were used for the purpose of evaluation of the system as listed in table 6.1. The first data set contains manually collected documents from the University of Waterloo various web sites, such as the Graduate Studies Office, Information Systems and Technology, Career Services, CO-operative Education, Health Services, and others. The data set also contains various documents from other Canadian web sites. The total number of documents in this set is 314 documents, and is categorized into 10 different categories (with some relevancy between the categories.) The second data set is a collection of Reuters news articles from the Yahoo! news site. The set contains 2340 documents classified into 20 different categories (with some relevancy between the categories as well.) The second data set was used by Boley et al in [7, 5, 6].

\(^1\)The authors mentioned here were contacted for obtaining a copy of their corpus but they informed us that the data is not made available for others
Table 6.1: Data Sets Descriptions

<table>
<thead>
<tr>
<th>Data Set</th>
<th>Description</th>
<th>Categories</th>
<th>Documents</th>
</tr>
</thead>
<tbody>
<tr>
<td>DS1</td>
<td>UW web site, Canadian web sites</td>
<td>10</td>
<td>314</td>
</tr>
<tr>
<td>DS2</td>
<td>Reuters news articles (from Yahoo! news)</td>
<td>20</td>
<td>2340</td>
</tr>
</tbody>
</table>

6.2 Effect of Phrase-based Similarity on Clustering Quality

In order to test the effectiveness of phrase matching in determining an accurate measure of similarity between documents, we conducted several experiments. The features extracted by the document analysis, the Document Index Graph model, the phrase matching algorithm, and the phrase-based similarity measure were evaluated. Significant improvement was achieved over using traditional individual-words representation and similarity measures.

The similarities calculated by our algorithm were used to construct a similarity matrix between the documents. We elected to use three standard document clustering techniques for testing the effect of phrase similarity on clustering [25]: (1) Hierarchical Agglomerative Clustering (HAC), (2) Single Pass Clustering, and (3) K-Nearest Neighbor Clustering (K-NN). For each of the algorithms, we constructed the similarity matrix and let the algorithm cluster the documents based on the presented similarity matrix.

In order to evaluate the quality of the clustering, we adopted two quality measures widely used in the text mining literature for the purpose of document clustering. The first is the F-measure, which combines the Precision and Recall ideas from the Information Retrieval literature. The second is Entropy, which provides a measure of "goodness" for un-nested clusters or for the clusters at one level of a hierarchical clustering. Entropy tells us how homogeneous a cluster is. (See section 2.3 for details about these evaluation measures.) Basically we would like to maximize the F-measure, and minimize the Entropy of clusters to achieve high quality clustering.

The F-measure is a better quality measure than Entropy for evaluating the clustering quality. Normally the Entropy measure will report a perfect cluster if the Entropy of the
cluster is zero (i.e. totally homogeneous). However, if a cluster contains all the documents from two different classes, its entropy will be zero as well. Hence entropy does not tell us if a cluster maps totally to one class or more, but the F-measure does. By using both measures we have confidence that our evaluation will be justified.

<table>
<thead>
<tr>
<th></th>
<th>Single-Term Similarity</th>
<th>Combined Similarity</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>F-measure</td>
<td>Entropy</td>
<td>F-measure</td>
</tr>
<tr>
<td>HAC</td>
<td>0.709</td>
<td>0.351</td>
<td>0.904</td>
</tr>
<tr>
<td>Single Pass</td>
<td>0.427</td>
<td>0.613</td>
<td>0.817</td>
</tr>
<tr>
<td>K-NN</td>
<td>0.228</td>
<td>0.173</td>
<td>0.834</td>
</tr>
</tbody>
</table>

\(^a\)Complete Linkage was used as the cluster distance measure for the HAC method since it tends to produce tight clusters with small diameter.

\(^b\)A document-to-cluster similarity threshold of 0.25 was used.

\(^c\)A \(K\) of 5 and a cluster similarity threshold of 0.25 were used.

**Table 6.2: Phrase-based Clustering Improvement**

The results listed in Table 6.2 show the improvement on the clustering quality using a combined similarity measure. The results were obtained using the first data set. The improvements shown were achieved at a similarity blend factor between 70% and 80% (phrase similarity weight — see section 4.4.1 for details on combining phrase-based similarity and single-term similarity.) The parameters chosen for the different algorithms were the ones that produced best results. The percentage of improvement ranges from 19.5% to 60.6% increase in the F-measure quality, and 9.1% to 46.2% drop in Entropy (lower is better for Entropy). It is obvious that the phrase based similarity plays an important role in accurately judging the relation between documents. It is known that Single Pass clustering is very sensitive to noise; that is why it has the worst performance. However, when the phrase similarity was introduced, the quality of clusters produced was pushed close to that produced by HAC and K-NN.

In order to better understand the effect of the phrase similarity on the clustering quality, we generated a clustering quality profile against the similarity blend factor.

Figure 6.1(a) illustrates the effect of introducing the phrase similarity on the F-measure of the resulting clusters. It is obvious that the phrase similarity enhances the F-measure of the clustering until a certain point (around a weight of 80%) and then its effect starts bringing down the quality. As we mentioned in section 4.4.1 that phrases alone cannot
capture all the similarity information between documents, the single-term similarity is still required, but to a smaller degree. The same can be seen from the Entropy profile in Figure 6.1(b), where Entropy is minimized at around 80% contribution of phrase similar-

Figure 6.1: Effect of Phrase Similarity on Clustering Quality
ity against 20% for the single-term similarity.

We demonstrated that phrase-based similarity is a more robust and accurate measure for clustering documents. We now shift focus to evaluating the incremental clustering algorithm introduced in Chapter 5.

6.3 Incremental Clustering

The evaluation of the proposed incremental clustering method is discussed here in terms of the quality of clustering. The method was tested using both the data sets described earlier.

Because the proposed method requires that every cluster be as cohesive as possible by ensuring that it keeps its pair-wise document similarity distribution high enough, the clusters formed by our method surpasses the other clustering methods in terms of the clustering quality.

We relied on all the quality measures described in Chapter 2, including the overall similarity in the clusters. We calculated the weighted average of the overall similarity as a measure of the coherency of the clusters. The parameters used for the different algorithms are the same as those used in the experiment described in section 6.2.

\[
\begin{array}{cccccc}
\text{Data Set 1} & & & & \text{Data Set 2} & \\
\text{Proposed Method} & 0.931 & 0.119 & 0.504 & 0.682 & 0.156 & 0.497 \\
\text{HAC} & 0.709 & 0.351 & 0.455 & 0.584 & 0.281 & 0.398 \\
\text{Single Pass} & 0.427 & 0.613 & 0.385 & 0.502 & 0.250 & 0.311 \\
\text{K-NN} & 0.228 & 0.173 & 0.367 & 0.522 & 0.161 & 0.452 \\
\end{array}
\]

\( ^a \text{Overall-Similarity} \)

Table 6.3: Proposed Clustering Method Improvement

Table 6.3 shows the result of the proposed clustering method against HAC, Single Pass, and K-NN clustering. For the small data set (1), the improvement was very significant, reaching over 70% improvement over k-NN (in terms of F-measure), 25% improvement over HAC, and 53% improvement over Single Pass. This is attributed to the fact
that the different categories of the documents do not have a great deal of overlap, which makes the algorithm able to avoid noisy similarities from other clusters.

For the second data set an improvement between 10% to 18% was achieved over the other methods. However, the F-measure was not really high compared to the first data set. By examining the actual documents and their classification it turns out that the documents do not have enough overlap in each single class, which makes it difficult to have an accurate similarity calculation between the documents. However, we were able to push the quality of clustering further by relying on accurate and robust phrase matching similarity calculation, and achieve higher clustering quality.

### 6.3.1 Evaluation of Document Re-assignment

Figure 6.2 shows the above mentioned results more clearly, showing the achieved improvement in comparison with the other methods. The figure shows also the effect of apply the re-assignment strategy discussed in section 5.2.3. The problem with incremental clustering is that documents usually do not end up where they should be. The re-assignment strategy we chose to use re-assigns documents that are seen as bad for some clusters to other clusters that can accept the document, all based on the idea of increasing the cluster similarity histogram. The re-assignment strategy showed a slight improvement over the same method without document re-assignment as shown in the figure.
Figure 6.2: Quality of Clustering Comparison
Conclusions and Future Research

7.1 Conclusions

This thesis involved a research on web mining, and in particular web content mining. Web mining involves a great deal of work in several areas, including data mining, text mining, pattern recognition, knowledge discovery, and sometimes artificial intelligence. We were mainly concerned with the problem of information categorization, and web document clustering in particular.

Challenged by a problem of finding inherent structure in web documents, and clustering the documents in a way that will make it easy to figure out the distinct topics that the documents talk about, we presented a framework for web document clustering based on a variety of ideas.

We presented a system composed of decoupled components for solving the problem. A web document analysis component that takes on the task of web document structure identification, and building structured documents out of the semi-structured web documents. Information in web documents does not lie in the content only, but in the inherent semi-structure of the web documents. By exploiting this structure we can achieve better clustering results.
Perhaps the most important component of this system, and the one that has most of the impact on performance, is the new document model introduced in this thesis, the Document Index Graph; which indexes web documents based on phrases and their levels of significance. The model enables us to perform phrase matching and similarity calculation between documents in a very robust and accurate way. The quality of clustering achieved using this model significantly surpasses the traditional vector space model based approaches.

Finally, we introduced an incremental document clustering algorithm based on a cohesiveness measure that was developed specifically to enhance the intra-cluster similarity, and thus increasing the quality of clustering. The algorithm is also capable of avoiding document insertion order problems, usually suffered by most incremental algorithms, by allowing a negotiation protocol between clusters to exchange documents that will improve their quality.

The problem of document categorization in general does not have one “good” solution. As reported by Mackassy et al [37], humans themselves produce different results if they are given a task to categorize documents manually. This essentially entails that automating the task of document categorization or clustering will not always be satisfying to all people. However, it is of our interest to create such automated systems to facilitate knowledge discovery, sharing, and exchange.

7.2 Future Research

The problem of information categorization has not been solved yet, and much more needs to be done. We are just scratching the surface in an effort to unveil the many correlated problems and issues involved.

The work that was presented in this thesis has been focused in two very sensitive areas in this line of research: (a) using informative features such as phrases in document similarity measurement, and (b) applying a coherency-guided approach to creating incremental clusters. The methods presented proved to be of higher efficiency than other methods, but they are by no means complete.

One direction that this work might continue on is to improve on the accuracy of
similarity calculation between documents by employing different similarity calculation strategies. Although the current scheme proved more accurate than traditional methods, there are still room for improvement.

Another extension to the approach of clustering presented here is to employ fuzzy logic into the process of clustering so as to fuzzify the membership of documents to several clusters. This essentially creates a foundation for allowing documents to belong to more than one cluster with varying degrees of memberships, and will reflect on the clustering process itself. We think that this approach is a very promising direction to work on.

Our intention for the continuation of this work is to build an Internet-enabled clustering system that has potentials for online daily usage. So far work has been on offline document corpora. The real use of the system should be through a working online model. Such online system has its own problems and issues other than the core clustering problem.

An interesting area that is developing nowadays is Web Services. A Web Service is a software entity deployed on a web server that could expose its functionality to others for use programmatically over the Internet. This kind of inter-object communication over the web has a very promising potential in the coming period, where it is getting adopted widely for its intuitive architecture and ease of use. The web document clustering system could be deployed in such a model as a web service acting as a back-end for others to consume over the Internet. If we could achieve such functionality and expose it to others, it could be easily adopted and improved continuously. If enough parties are interested it could also be commercialized.

The problem could be extended to the paradigm of Internet Agents, where intelligent agents will collaborate on performing different tasks (either on the same problem with different approaches, or on different problems) and organize their efforts in order to achieve the highest degree of performance in knowledge discovery.

What may sound today as a dream, tomorrow it could be a reality.
Implementation

The proposed system is implemented as four subsystems:

1. The **Document Structure Identification** subsystem,
2. The **Document Index Graph** subsystem,
3. The **Matching and Similarity Calculation** subsystem, and
4. The **Clustering** subsystem.

Figure A.1 illustrates the system architecture. The various components involved in each subsystem are shown, and the integration of the subsystems is illustrated.
Figure A.1: System Architecture
The system was implemented in the C++ language. There are 40 core classes, and 10 helper and user interface classes, with a total of around 9,700 lines of code. The project was built using Microsoft Visual C++; initially using version 6.0, then shifted to version 7.0 to make use of the new ATL classes. Libraries used include MFC, ATL, and STL. All classes were built from scratch, except for the Porter stemmer and stop word removal class. The Porter algorithm is available publicly [41]. Table A.1 lists the core classes along with a brief description of each class.

Table A.1: Classes Description

<table>
<thead>
<tr>
<th>Class</th>
<th>Parent Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Document Structure Identification</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parser</td>
<td></td>
<td>HTML document scanner and parser</td>
</tr>
<tr>
<td>DocCleaner</td>
<td></td>
<td>Whitespace removing, document cleaning</td>
</tr>
<tr>
<td>Markup</td>
<td></td>
<td>Core HTML structure identification logic</td>
</tr>
<tr>
<td>HTMLTag</td>
<td></td>
<td>Helper HTML tag class</td>
</tr>
<tr>
<td>TextBoundary</td>
<td></td>
<td>Basic text boundary analysis</td>
</tr>
<tr>
<td>SentenceBoundary</td>
<td>TextBoundary</td>
<td>Sentence boundary detector</td>
</tr>
<tr>
<td>WordBoundary</td>
<td>TextBoundary</td>
<td>Word boundary detector</td>
</tr>
<tr>
<td>FiniteStateMachine</td>
<td></td>
<td>For boundary detectors lexical analysis</td>
</tr>
<tr>
<td>Porter</td>
<td></td>
<td>Word stemming, stop word removal</td>
</tr>
<tr>
<td>XmlDocument</td>
<td></td>
<td>XML document (using our own DTD)</td>
</tr>
<tr>
<td>XmlDoc</td>
<td></td>
<td>Basic XML operations</td>
</tr>
<tr>
<td>XmlDocument</td>
<td>XmlDocument</td>
<td>For outputting XML to files</td>
</tr>
<tr>
<td>XmlWriter</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Document Index Graph

<table>
<thead>
<tr>
<th>Class</th>
<th>Parent Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WordGraph</td>
<td></td>
<td>Core word graph representation</td>
</tr>
<tr>
<td>GraphNode</td>
<td></td>
<td>Graph node class</td>
</tr>
<tr>
<td>GraphEdge</td>
<td></td>
<td>Graph edge class</td>
</tr>
<tr>
<td>NodeInfo</td>
<td></td>
<td>Stores node information</td>
</tr>
<tr>
<td>Class</td>
<td>Parent Class</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>--------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>DocTable</td>
<td></td>
<td>Stores document table of a graph node</td>
</tr>
<tr>
<td>DocTableEntry</td>
<td></td>
<td>Document entry in DocTable</td>
</tr>
<tr>
<td>EdgeTable</td>
<td></td>
<td>Stores edge info for a document entry</td>
</tr>
<tr>
<td>EdgeTableEntry</td>
<td></td>
<td>Edge entry in EdgeTable</td>
</tr>
<tr>
<td>SentenceEntry</td>
<td></td>
<td>Sentence entry in EdgeTableEntry</td>
</tr>
</tbody>
</table>

**Matching and Similarity Calculation**

<table>
<thead>
<tr>
<th>Class</th>
<th>Parent Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XmlParser</td>
<td>XmlDocument</td>
<td>For parsing XML documents</td>
</tr>
<tr>
<td>XmlDocFeeder</td>
<td></td>
<td>Feeds sentences to the matching engine</td>
</tr>
<tr>
<td>DscEngine</td>
<td></td>
<td>Core doc. structure-based matching engine</td>
</tr>
<tr>
<td>SimilarityMatrix</td>
<td></td>
<td>Document similarity information</td>
</tr>
<tr>
<td>DocSim</td>
<td></td>
<td>Similarity between one doc. and others</td>
</tr>
<tr>
<td>SimTable</td>
<td></td>
<td>Temporary current similarity info</td>
</tr>
<tr>
<td>SimEntry</td>
<td></td>
<td>One entry in SimTable</td>
</tr>
</tbody>
</table>

**Clustering**

<table>
<thead>
<tr>
<th>Class</th>
<th>Parent Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClusterMgr</td>
<td></td>
<td>Base cluster manager</td>
</tr>
<tr>
<td>Cluster</td>
<td></td>
<td>Base cluster class</td>
</tr>
<tr>
<td>DscClusterMgr</td>
<td>ClusterMgr</td>
<td>Doc. structure-based cluster mgr.</td>
</tr>
<tr>
<td>DscCluster</td>
<td>Cluster</td>
<td>Doc. structure-based cluster</td>
</tr>
<tr>
<td>AhcClusterMgr</td>
<td>ClusterMgr</td>
<td>Agglom. Hierarchical Clustering mgr.</td>
</tr>
<tr>
<td>AhcCluster</td>
<td>Cluster</td>
<td>Agglom. Hierarchical Clustering cluster</td>
</tr>
<tr>
<td>SinglePassCluster</td>
<td>Cluster</td>
<td>Single Pass cluster</td>
</tr>
<tr>
<td>KnnCluster</td>
<td>Cluster</td>
<td>K-Nearest Neighbor cluster</td>
</tr>
<tr>
<td>DocClassMgr</td>
<td></td>
<td>Document class manager</td>
</tr>
<tr>
<td>DocClass</td>
<td></td>
<td>Document class</td>
</tr>
</tbody>
</table>
The following is a listing of some of the core classes mentioned above: **Markup**, DscEngine, WordGraph, SimTable, SingleMatchEntry, SimSentence, SimEntry, ClusterMgr, Cluster, DscClusterMgr, and DscCluster.

```cpp
// Markup.h: interface for the CMarkup class.

#include "afxtempl.h"
#include "HTM.LTag.h"
#include "XMLDoc.h"
#include "DocCleaner.h"

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

enum HTMLContext {
};

class CScanner;
class CHTM.LTag;
class CMarkup {
public:
    void EndMarkup();
    void SetScanner(CScanner* pScanner);
    void EndTag(CTM.LTag tag);
    void BeginTag(CTM.LTag tag);
    CMarkup();
    virtual ~CMarkup();

    void SetXMLDoc(CXMLDoc* pdocXML)
    {
        ASSERT(pdocXML);
        m_pXML = pdocXML;
    }
    void Reset(void);
    void OnMetaTag(const CXMLTag& tagMeta);
```
protected:
    unsigned long m_nStartSectionID;
    unsigned long m_nEndSectionID;
    unsigned long m_nStartBlockID;
    unsigned long m_nEndBlockID;
    void EmitTaggedText(CTag tag, CString& strText);
    void EmitSectionText(CString& strText);
    CList<CTag, CTag>& m_TagList;
    CScanner* m_pScanner;
    CXMDoc* m_pDocXML;
    CDocCleaner m_Cleaner;
    bool m_bParsing;
    bool m_bBodyStarted;
    bool m_bHeadEnded;
    bool m_bInsideSection;
};

#endif // !defined(AFX_MARKUP_H__C6D90EE0_E04A_47DB_96E9_989C00C4753E__INCLUDED_

// Markup.cpp: implementation of the CMarkup class.
//
#if _DEBUG
#include "stdafx.h"
#endif

#include "DSC.h"
#include "Markup.h"

#include "afxtempl.h"

#include "Scanner.h"
#include "HTMLTag.h"
#include "DocCleaner.h"

#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[]=__FILE__;
#define new DEBUG_NEW
#endif

// Construction/Destruction

Derived from CXMLDoc:

    // Constructor
    public:
        CMarkup();
        ~CMarkup();

    // Destructor
    public:
        virtual void ~CMarkup();

    // Method to test
    public:
        CMarkup(const CMarkup&);
CMarkup::CMarkup()
: m_bParsing(false)
, m_pScanner(NULL)
, m_bHeadEnded(false)
, m_bBodyStarted(false)
, m_nStartSectionID(0)
, m_nEndSectionID(0)
, m_nStartBlockID(0)
, m_nEndBlockID(0)
, m_pdocXML(NULL)
, m_bInsideSection(false)
{
}

CMarkup::~CMarkup()
{
}

void CMarkup::BeginTag(CTag tag)
{
    ELEMENT_TAG_ID tagType = tag.GetTagType();
    if (tagType == TAGID_UNKNOWN) // unknow tag
        return; // ignore

    // add tag to the list of open tags
    m_TagList.AddHead(tag);

    CString strTextBlock;

    switch (tagType) {
    case TAGID_BODY:
        // a <BODY> tag denotes a new text section start
        m_bBodyStarted = true;
        m_nStartSectionID = tag.GetID();
        m_nStartBlockID = m_nStartSectionID;
        if (!m_bInsideSection) {
            m_pdocXML->BeginSection();
            m_bInsideSection = true;
        }
        break;

    case TAGID_BR: // Line Break
    case TAGID_P: // Paragraph
    case TAGID_LI: // List Item
    case TAGID_TABLE: // Table
case TAGID_TD:  // Table Data
    case TAGID_HR:  // Horizontal Rule
        // new block boundary, emit previous block
        m_nEndBlockID = tag.GetID();
        strTextBlock = m_pScanner->GetTaggedText(m_nStartBlockID,
                                                m_nEndBlockID);
        EmitSectionText(strTextBlock);
        // set new block start to current tag
        m_nStartBlockID = m_nEndBlockID;
        break;

    case TAGID_H1:
    case TAGID_H2:
    case TAGID_H3:
    case TAGID_H4:
    case TAGID_H5:
    case TAGID_H6:
        // a <H?> tag denotes a new text section start
        m_nEndSectionID = tag.GetID();
        m_nEndBlockID = m_nEndSectionID;
        strTextBlock = m_pScanner->GetTaggedText(m_nStartBlockID,
                                                m_nEndBlockID);
        EmitSectionText(strTextBlock);
        // a new <H?> tag denotes a new section, so end current section
        // (new section body will start after the closing </H?> tag is received)
        if (m_bInsideSection) {
            m_pDocXML->EndSection();
            m_bInsideSection = false;
        }
        break;

    case TAGID_META:
        OnMetaTag(tag);
        break;
    }
}

void CMarkup::EndTag(CTHTMLTag tag)
{
    CString strTaggedText;
    ELEMENT_TAG_ID tagType = tag.GetTagType();
    if (tagType == TAGID_UNKNOWN) // unknown tag
        return;  // ignore
// locate the corresponding start tag in the tag list
CHTMLTag startTag;
POSITION pos, pos2;
pos = m_TagList.GetHeadPosition();
if (pos == NULL)
    return;

BOOL bFound = FALSE;
while (pos != NULL && !bFound) {
pos2 = pos;
    startTag = m_TagList.GetNext(pos);
    if (tag.IsMatching(startTag)) {
        m_TagList.RemoveAt(pos2);
        bFound = TRUE;
    }
}

ASSERT(m_pScanner);

// retrieve the text contained between the start and end tag from the scanner
if (bFound) {
    strTaggedText = m_pScanner->GetTaggedText(startTag.GetID(), tag.GetID());
}

CSTRING strTextBlock;
switch (tagType) {
    case TAGID_TITLE:
        if (!strTaggedText.IsEmpty()) {
            EmitTaggedText(startTag, strTaggedText);
        }
        break;
    case TAGID_HEAD: // just in case the document doesn’t have a <BODY> tag!
        m_bHeadEmded = true;
        m_nStartSectionID = tag.GetID();
        m_nStartBlockID = m_nStartSectionID;
        if (!m_bInsideSection) {
            m_pdocXML->BeginSection();
            m_bInsideSection = true;
        }
        break;
    case TAGID_BR: // Line Break
    case TAGID_P: // Paragraph
    case TAGID_LI: // List Item
    case TAGID_TABLE: // Table
case TAGID_TD:    // Table Data
    // block closing boundary, emit previous block
    m_nEndBlockID = tag.GetID();
    strTextBlock = m_pScanner->GetTaggedText(m_nStartBlockID, m_nEndBlockID);
    EmitSectionText(strTextBlock);
    // set new block start to current tag
    m_nStartBlockID = m_nEndBlockID;
    break;
    case TAGID_H1:
    case TAGID_H2:
    case TAGID_H3:
    case TAGID_H4:
    case TAGID_H5:
    case TAGID_H6:
        // a closing </H?> tag denotes closing a section heading, so start a
        // new section body
        if (!m_bInsideSection) {
            m_pdocXML->BeginSection();
            m_bInsideSection = true;
        }
        m_nStartSectionID = tag.GetID();
        m_nStartBlockID = m_nStartSectionID;
        if (!strTaggedText.IsEmpty()) {
            EmitTaggedText(startTag, strTaggedText);
        }
        break;
    case TAGID_HTML:
        // if we receive a closing </HTML> tag, end parsing
        m_bParsing = false;
        break;
    }
}

void CMarkup::EmitTaggedText(CHTMLTag tag, CString& strText)
{
    // clean text before sending to xml output
    m_Cleaner.TranslateCharacters(strText);
    m_Cleaner.RemoveWhiteSpace(strText);

    ELEMENT_TAG_ID tagType = tag.GetTagType();

    switch (tagType) {
        case TAGID_TITLE:
            m_pdocXML->SetTitle(strText);
            break;
        // other cases
    }
break;
case TAGID_H1:
case TAGID_H2:
case TAGID_H3:
case TAGID_H4:
case TAGID_H5:
case TAGID_H6:
    m_pdocXML->AddSectionHeading(strText);
    break;
}
}

void CMarkup::EmitSectionText(CString& strText)
{
    // clean text before sending to xml output
    m_Cleaner.TranslateCharacters(strText);
    m_Cleaner.RemoveWhiteSpace(strText);

    if (strText.IsEmpty())
        return;

    m_pdocXML->AddSectionText(strText);
}

void CMarkup::EndMarkup()
{
    // emit last section text
    CString strTextBlock;
    strTextBlock = m_pScanner->GetTaggedText(m_nStartBlockID, -1);
    EmitSectionText(strTextBlock);
    m_pdocXML->EndSection();
}

void CMarkup::SetScanner(CScanner* pScanner)
{
    m_pScanner = pScanner;
}

void CMarkup::Reset(void)
{
    m_bParsing = false;
    m_bHeadEnded = false;
    m_bBodyStarted = false;
    m_bInsideSection = false;
    m_TagList.RemoveAll();
m_nStartSectionID = 0;
m_nEndSectionID = 0;
m_nStartBlockID = 0;
m_nEndBlockID = 0;
m_pdocXML = NULL;
}

void CMarkup::OnMetaTag(const CHTMLTag& tagMeta)
{
    CString attrValue = tagMeta.GetAttribute(_T("name"));
    if (attrValue.IsEmpty())
        return;

    attrValue.MakeLower();
    if (attrValue == _T("description")) {
        m_pdocXML->SetDescription(tagMeta.GetAttribute(_T("content")));
    }
    else if (attrValue == _T("keywords")) {
        m_pdocXML->SetKeywords(tagMeta.GetAttribute(_T("content")));
    }
}

// DSCEngine.h: interface for the CDSCEngine class.
//
// //////////////////////////////////////////////////////////////////////////

#ifdef AFX_DSCENGINE_H__710324FF_94C6_4938_8999_9D5DACF4638D__INCLUDED_
#define AFX_DSCENGINE_H__710324FF_94C6_4938_8999_9D5DACF4638D__INCLUDED_
#endif

#include "SimTable.h" // Added by ClassView
#include "DocInfo.h" // Added by ClassView
#include "WordGraph.h" // Added by ClassView
//#include "XMLDoc.h"

class CXMLDoc;

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include "afx.h"
#include "atlcoll.h"

class CDocSim;
class ClusterMgr;
typedef void (*DSC_ENGINE_CALLBACK)(LPVOID, DWORD);

class CDSCEngine
{
public:
    CDSCEngine();
    virtual ~CDSCEngine();
    void ResetState();
    void BeginDocument(int nDocID);
    int ProcessSentence(const CString &strSentence, DSC_SENTENCEINFO sntInfo);
    void DumpGraph();
    void ProcessDocument(CXMLDoc& doc);

    enum CLUSTERING_METHOD {
        DSC = 0, AHC = 1, SINGLE_PASS = 2, KNN = 3
    };

    enum DSC_ENGINE_OPTION {
        UPDATE_SIMILARITY = 0x0001,
        REPORT_PROGRESS = 0x0002,
        REPORT_STATS = 0x0004,
        STEM_WORDS = 0x0008,
        MATCH_SINGLE_WORDS = 0x0010,
        OUTPUT_SIM_MATRIX = 0x0020,
        REMOVE_STOP_WORDS = 0x0040,
        KEEP_PHRASE_STOP_WORDS = 0x0080,
        SKIP_NUMERIC_WORDS = 0x0100,
        SKIP_ALLSTOPWORDS PHRASES = 0x0200,
    };

    void SetProgressCallback(DSC_ENGINE_CALLBACK pfCallback, DWORD dwCookie);
    void SetStatsCallback(DSC_ENGINE_CALLBACK pfnCallback, DWORD dwCookie);
    void DumpSimTable(void);
    bool OutputSimTable(const CString& sFileName);
    void SetOptions(unsigned int uOptions);
    CString GetNextWord(const CString& strSentence, int& pos);

    void UpdateSimMatrixFile(void);
    bool Initialize(void);
    bool Finalize(void);
    void CalcDocSim(CDocSim* pDocSim);
    void SetSimBlendFactor(double dBlendFactor);
    CString StemWord(const CString& strWord);

    void SetClusteringMethod(CDSCEngine::CLUSTERING_METHOD method)
    { m_ClusteringMethod = method; }
protected:
    void BreakString(const CString& strString);
    void UpdateSimilarity(int nDocID, CGraphNode* pPrev, CGraphNode* pCurrent,
                           int nEdgeID, DSC_SIGLEVEL sigLevel);

union tagDSC_OPTIONS {
    struct {
        unsigned UpdateSim : 1;
        unsigned ReportProgress : 1;
        unsigned ReportStats : 1;
        unsigned StemWords : 1;
        unsigned MatchSingleWords : 1;
        unsigned OutputSimMatrix : 1;
        unsigned RemoveStopWords : 1;
        unsigned KeepPhraseStopWords : 1;
        unsigned SkipNumericWords : 1;
        unsigned SkipAllStopWordsPhrases : 1;
    };
    unsigned All;
} m_Options;

CLUSTERING_METHOD m_ClusteringMethod;

int m_iCurrDoc;
DSC_DOCINFO m_CurrentDoc;
DSC_SIGLEVEL m_CurrentLevel;
DSC_CLUSTERING_STATS m_Stats;
CWordGraph m_WordGraph;
CStringArray m_saCurrentString;
CSimTable m_SimTable;
DSC_ENGINE_CALLBACK m_pfnReportProgress;
DSC_ENGINE_CALLBACK m_pfnReportStats;
int m_nReportFreq;
DWORD m_dwProgressCookie;
DWORD m_dwStatsCookie;
CString m_strCurrentString;
CStdioFile m_fileSimMatrix;
// the norm of each document <docID, normVal> (according to phrase equation)
CAtlMap<unsigned int, unsigned int> m_DocNorm;
double m_dSingleWordNorm;
// norm of single words
CAtlMap<unsigned int, double> m_DocSingleNorm;
double m_dSimBlendFactor;
ClusterMgr* m_pClusterMgr;
Implementation

public:

    void SetNumClusters(int num)
    {  m_nNumClusters = num;
    }
protected:
    int m_nNumClusters;
public:

    void SetClusterMgr(ClusterMgr* pClusterMgr)
    {  m_pClusterMgr = pClusterMgr;
    }
    bool OutputSimMatrix(CStdioFile& fileout);
protected:
    // minimum word count in a document to be processed
    int m_nMinWordCount;
public:

    void SetMinWordCount(int nMinCount)
    {  m_nMinWordCount = nMinCount;
    }
};
#endif  // !defined(
    AFX_DSCENGINE_H__710324FP_94C6_4938_8999_9D5D4CF4638D__INCLUDED_)

// DSCEngine.cpp: implementation of the CDSCEngine class.
//
#line 140 "DSCEngine.h"

#include "stdafx.h"
#include "DSC.h"
#include "DSCEngine.h"
#include "math.h"

// dependencies
#include "XMLDoc.h"
#include "GraphNode.h"
#include "XMLDocFeeder.h"
#include "Porter.h"
#include "DocSim.h"
#include "ClusterMgr.h"

extern CDSCApp theApp;
```c
#define _DEBUG
#undef THIS_FILE
static char THIS_FILE[]="__FILE__;
#define new DEBUG_NEW
#endif

// define the separator character between words in a single phrase
#define WORD_SEPARATOR_CHAR _T(\"\")

CDSCEngine::CDSCEngine()
: m_pfnReportProgress(NULL)
, m_pfnReportStats(NULL)
, m_dwProgressCookie(0)
, m_dwStatsCookie(0)
, m_iCurrDoc(0)
, m_nReportFreq(20)
, m_strCurrentString(_T("\n"))
, m_sSingleWordNorm(0.0)
, m_sSimBlendFactor(0)
, m_ClusteringMethod(CDSCEngine::DSC)
, m_nNumClusters(0)
, m_pClusterMgr(NULL)
, m_nMinWordCount(0)
{
    ZeroMemory(&m_Options, sizeof(m_Options));
    ZeroMemory(&m_Stats, sizeof(m_Stats));
    // initialize variables
    ResetState();
}

CDSCEngine::~CDSCEngine()
{
}

void CDSCEngine::ResetState()
{
    m_CurrentDoc.nDocID = -1;
    m_CurrentDoc.nWordCount = 0;
    m_CurrentDoc.nUniqueWordCount = 0;
```
m_CurrentLevel = SIG_UNKNOWN;

m_pfnReportProgress = NULL;
m_dwProgressCookie = 0;
m_pfnReportStats = NULL;
m_dwStatsCookie = 0;
m_iCurrDoc = 0;
m_nReportFreq = 20;
ZeroMemory(&m_Options, sizeof(m_Options));
m_strCurrentString = _T(""); 
m_saCurrentString.RemoveAll();
m_SimTable.RemoveAll();
ZeroMemory(&m_Stats, sizeof(m_Stats));
m_WordGraph.Clear();
m_DocNorm.RemoveAll();
m_DocSingleNorm.RemoveAll();
if (m_pClusterMgr)
  m_pClusterMgr->RemoveAll();
}

void CDSCEngine::BeginDocument(int nDocID)
{
  m_CurrentDoc.nDocID = nDocID;
}

// sentence level processing
// return number of words found in sentence
int CDSCEngine::ProcessSentence(const CString &strSentence, DSC_SENTENCEINFO
  sntInfo)
{
  // make sure we don’t have an empty string
  if (strSentence.IsEmpty())
    return 0;

  // first, break down the string into words (result is stored in a member
  CStringArray)
  int currPos = 0;
  CString strWord = GetNextWord(strSentence, currPos);
  CString strOriginal = strWord;
  string s_word;
  if (m_Options.StemWords) {
    s_word = theApp.m_Stemmer.stripAffixes(string(strOriginal));
    strWord = s_word.c_str();
  }
```cpp
int nWords = 0;
int iSeq = 0;
// process each word
CGraphNode* pNode = NULL;
CGraphNode* pPrevNode = NULL;
while (strWord != "") {
    // add a node for the current word (if the node already exists nothing happens)
    long nodeCount = m_WordGraph.GetNodeCount();
pNode = m_WordGraph.AddNode(strWord);
    bool bNew = (nodeCount != m_WordGraph.GetNodeCount());

    if (bNew && theApp.m_ Stemmer. isStopWord(string(strOriginal))) // check for stop-word
        pNode->SetStopWord();
    if (CDocCleaner:: IsNumeric(strOriginal)) // check for numeric-word
        pNode->SetNumeric();

    if (m_Options. MatchSingleWords) {
        bool bAdd = false;
        if (m_Options. RemoveStopWords)
            if (!pNode->IsStopWord())
                bAdd = true; // keep record of matching single words
        else
            bAdd = true;

        if (bAdd && m_Options. SkipNumericWords) // check if word is numeric
            if (pNode->IsNumeric())
                bAdd = false;

    if (bAdd) { // word is neither a stop-word or numeric (or those options were not specified)
        m_SimTable. AddSingleMatch(pNode);
        // update single-word norm
        int dx = sntInfo. sigLevel;
        if (nodeCount == m_WordGraph. GetNodeCount()) { // word already present, check if this doc has an entry
            DSC_TERMFREQ* pTFreq = pNode->GetTermFreq(m_iCurrDoc);
            if (pTFreq) // this doc has an entry -> update norm using incremental equation
                m_dSingleWordNorm += 2*(pTFreq->nVeryHigh*SIG_VERY_HIGH + pTFreq->nHigh*SIG_HIGH + pTFreq->nMedium*SIG_MEDIUM + pTFreq->nLow*SIG_LOW)*dx + dx*dx;
        else
```
m_dSingleWordNorm += dX*dX; // no entry for this doc ->
  add new term to norm
}
else
  m_dSingleWordNorm += dX*dX; // new unique word -> add new
  term to norm
}
}
pNode->UpdateFrequency(sntInfo.nDocID, sntInfo.sigLevel);
// check if previous node exists
int nEdgeID = -1;
if (pPrevNode) {
  nEdgeID = pPrevNode->IsLinkedTo(pNode);
  if (nEdgeID == -1) // check if previous node already points to current
    node
    nEdgeID = m_WordGraph.LinkNodes(pPrevNode, pNode); // if they
    are not linked, link them
  // update the previous node’s info to reflect a sentence continuation
  sntInfo.nSeq = iSeq++;
  pPrevNode->AddSentenceEntry(sntInfo, nEdgeID);

  if (m_Options.UpdateSim & UPDATE_SIMILARITY) {
    // update similarity table
    UpdateSimilarity(sntInfo.nDocID, pPrevNode, pNode, nEdgeID,
    sntInfo.sigLevel);
  }
}
nWords++;
strOriginal = strWord = getNextWord(strSentence, currPos);
if (m_Options.StemWords) {
  s_word = theApp.m_Stemmer.stripAffixes(string(strOriginal));
  strWord = s_word.c_str();
}
pPrevNode = pNode; // move to new node
return nWords;
}

CString CDSCEngine::GetNextWord(const CString& strSentence, int& pos)
{
  CDSCApp* pApp = (CDSCApp*)AfxGetApp();
  string s_word;

  CString strWord = strSentence.Tokenize(WORD_SEPARATOR_CHAR, pos);
if (strWord == "")
    return strWord;

   // skip over stop-words (if options specify so)
if (m_Options.RemoveStopWords && !m_Options.KeepPhraseStopWords) {
    while (strWord != "" && pApp->m_Stepmer.isStopWord(string(strWord)))
        strWord = strSentence.Tokenize(WORD_SEPARATOR_CHAR, pos);
    if (strWord == "")
        return strWord;
}

    return strWord.MakeLower();

    if (m_Options.StemWords) {
        // look for the first non-empty stemmed word
        CStrin strTemp = strWord;
        while (strTemp != "" && s_word == "") {
            strTemp = strSentence.Tokenize(WORD_SEPARATOR_CHAR, pos);
            if (strTemp != "")
                s_word = pApp->m_Stepmer.stripAffixes(string(strTemp));
            else
                s_word = "";
        }
        if (strWord == "")? "" : s_word.c_str();
    }
    return strWord;
}

// Breaks down the given string into words and puts them in a member
CStrin Array
void CDSCEngin::BreakString(const CStrin &strString)
{
    int iStart = 0, iEnd = 0;
    int nLen = strString.GetLength();
    CStrin strWord;

    // clean up CStrinArray for the new words
    m_saCurrentString.RemoveAll();

    // if string is empty do nothing
    if (strString.IsEmpty())
        return;

    // Note: only single separator character is allowed between words
    while (iEnd < nLen) {
iEnd = strString.Find(WORD_SEPARATOR_CHAR, iStart); // locate the end of
the current word
if (iEnd == -1) // no more separators
   iEnd = nLen;
strWord = strString.Mid(iStart, iEnd - iStart);
iStart = iEnd + 1;
// add the current word to the array
m_saCurrentString.Add(strWord);
}
}
void CDSCEngine::DumpGraph()
{
#ifdef _DEBUG
   m_WordGraph.Dump(afxDump);
#endif
}
void CDSCEngine::UpdateSimilarity(int nDocID, CGraphNode *pPrev, CGraphNode *pCurrent, int nEdgeID, DSC_SIGLEVEL sigLevel)
{
   DSC_SIMSENTENCE sntSim;
   DSC_SENTENCEINFO sntInfo;
   BOOL bMore = FALSE;

   // detect two-word or more similarities with other documents in the context
   // of the current sentence
   if (pPrev) {
      // get first matching sentence entry from previous node's info table
      bMore = pPrev->GetFirstSentence(nDocID, &sntInfo, nEdgeID);
      while (bMore) {
         sntSim.nSID = sntInfo.nSID;
         sntSim.nSeq = sntInfo.nSeq;
         sntSim.sigLevelX = sigLevel; // our sig level
         sntSim.sigLevelY = sntInfo.sigLevel; // matching phrase sig level
         // update similarity table with the found matching sentence entry
         m_SimTable.Update(sntInfo.nDocID, sntSim, pPrev, pCurrent);
         // get next matching sentence entry
         bMore = pPrev->GetNextSentence(nDocID, &sntInfo, nEdgeID);
      }
   }
}
void CDSCEngine::ProcessDocument(CXMLDoc &doc)
{
XMLDocFeeder xmlFeeder;
xmlFeeder.SetXMLDoc(&doc);
m_iCurrDoc = doc.GetDocID();
CDocSim docSim(m_iCurrDoc);

int nWordCount = doc.GetWordCount();
if (nWordCount < m_nMinWordCount) // skip documents that don't have enough words
    return;

int nCurrWords = 0;
int nOldNodes = m_STATS.nGraphNodes;
int nOldEdges = m_STATS.nGraphEdges;
int nChunk = (int)((nWordCount * m_nReportFreq/100.0);
m_dSingleWordNorm = 0.0;

m_SimTable.RemoveAll();
m_SimTable.SetDocID(m_iCurrDoc);

// process document here

// calculate doc norm: summation of (sentence lengths * siglevel)
unsigned int naLevelWordCount[SIG_COUNT];
xmlFeeder.GetLevelWordCount(naLevelWordCount);
unsigned int nNorm = 0;
for (int i=0; i<SIG_COUNT; i++)
    nNorm += naLevelWordCount[i] * i;
m_DocNorm[m_iCurrDoc] = nNorm; // store it in the docNorm map

DSC_SENTENCE snt;
DSC_SENTENCEINFO sntInfo;
while (xmlFeeder.GetNextSentence(&snt)) {
    sntInfo = snt.sntInfo;
    sntInfo.nDocID = m_iCurrDoc;
    int nWords = ProcessSentence(snt.strSnt, sntInfo);
    nCurrWords += nWords;

    // report progress
    if (m_Options.ReportProgress && m_pfnReportProgress && nCurrWords>nChunk)
    {
        int nPercent = (int)(((double)nCurrWords/nWordCount)*100);
        (*m_pfnReportProgress)((LPVOID)&nPercent, m_dwProgressCookie);
        nChunk = nChunk;
    }
}
if (m_Options.ReportProgress && m_pfnReportProgress) {
    int nPercent = 100;
    (*m_pfnReportProgress)((LPVOID)&nPercent, m_dwProgressCookie);
}

// report statistics
if (m_Options.ReportStats && m_pfnReportStats) {
    // update statistics
    m_Stats.nGraphNodes = m_WordGraph.GetNodeCount();
    m_Stats.nGraphEdges = m_WordGraph.GetEdgeCount();
    m_Stats.nLastDocNodes = m_Stats.nGraphNodes - mOldNodes;
    m_Stats.nLastDocEdges = m_Stats.nGraphEdges - mOldEdges;
    (*m_pfnReportStats)((LPVOID)&m_Stats, m_dwStatsCookie); // call stats
    callback function
}

m_SimTable.MergeDuplicates();

m_DocSingleNorm[m_iCurrDoc] = m_dSingleWordNorm;

if (m_Options.OutputSimMatrix) { // output similarity matrix to file
    if (m_fileSimMatrix.m_pStream == NULL)
        if (!m_fileSimMatrix.Open("simmatrix.txt", CFile::modeCreate | CFile::modeWrite | CFile::typeText))
            AfxMessageBox("Cannot create SimilarityMatrix output file.");
}

CalcDocSim(&docSim); // calculate doc similarity to previous documents

CDSCApp* pApp = (CDSCApp*)AfxGetApp();

// different clustering methods are handled with different cluster managers
// send doc sim table to the appropriate cluster manager to process it
if (m_pClusterMgr)
    m_pClusterMgr->ProcessDocument(m_iCurrDoc);
}

void CDSCEngine::SetProgressCallback(DSCENGINE_CALLBACK pfnCallback, DWORD dwCookie)
{
    m_pfnReportProgress = pfnCallback;
    m_dwProgressCookie = dwCookie;
}
void CDSCEngine::SetStatsCallback(DSC_ENGINE_CALLBACK pfnCallback, DWORD dwCookie)
{
    m_pfnReportStats = pfnCallback;
    m_dwStatsCookie = dwCookie;
}

bool CDSCEngine::OutputSimTable(const CString& sFileName)
{
    CStdioFile outfile;
    if (!outfile.Open(sFileName, CFile::modeCreate | CFile::modeWrite))
        return false;

    int nSimDocCount = m_SimTable.GetDocCount();
    CString strLine2; strLine2.Preallocate(256);
    strLine2.Format(_T("SimTable:\d\n"), m_iCurrDoc);
    if (nSimDocCount == 0) {
        strLine2 += _T("Empty\n");
        outfile.WriteString(strLine2);
        outfile.Close();
        return true;
    }
    else {
        CTemp strTemp;
        strTemp.Format(_T("%d\n"), nSimDocCount);
        strLine2 += strTemp;
        outfile.WriteString(strLine2);
    }

    CGraphNode* pNode = NULL;
    for (int iDoc = 0; iDoc < nSimDocCount; iDoc++) {
        CSimEntry* pSimEntry = m_SimTable.GetSimEntry(iDoc);
        int nDocID = pSimEntry->nDocID;
        strLine2.Format(_T("DOC:\d\n"), nDocID);
        outfile.WriteString(strLine2);

        // get phrase matches
        int nSnt = pSimEntry->GetSentenceCount();
        strLine2.Format(_T("\n"), nSnt);
        outfile.WriteString(strLine2);
        CSimSentence* pSimSnt = NULL;
        CString strSimSnt;
        for (int j = 0; j < nSnt; j++) {
            pSimSnt = pSimEntry->GetSimSentence(j);
        }
    }
}
// skip sentences that are all-stop-words
if (m_Options.SkipAllStopWordsPhrases && pSimSnt->IsStopWords())
    continue;
strSimSnt.Preallocate(128);
pSimSnt->GetSentence(strSimSnt);
strLine2.Format(_T("%d:%.1f,%d:%.1f
"), pSimSnt->nCountX, pSimSnt->fWeightX, pSimSnt->nCountY, pSimSnt->fWeightY, strSimSnt);
outfile.WriteString(strLine2);
}
outfile.WriteString(_T("\n"));

// get single word matches
CSingleMatchEntry* pSingleMatch = m_SimTable.GetSingleMatchEntry(nDocID);
if (pSingleMatch) {
    CString strWord;
    int nSingleCount = pSingleMatch->GetMatchCount();
    strLine2.Format(_T("%d single matches: %d\n"), nSingleCount);
    outfile.WriteString(strLine2);
    if (nSingleCount) {
        for (int i=0; i<nSingleCount; i++) {
            pNode = pSingleMatch->GetNode(i);
            strWord = pNode->GetWord();
            outfile.WriteString(strWord);
            if (i<nSingleCount-1) outfile.WriteString(_T(",
"));
        }
    }
    outfile.WriteString(_T("\n"));
}
return true;

void CDSCEngine::SetOptions(unsigned int uOptions)
{
    m_Options.All |= uOptions;
}

void CDSCEngine::UpdateSimMatrixFile(void)
{
    #define MAX_MATCH_LENGTH_BINS 100

    int histogram[MAX_MATCH_LENGTH_BINS];
    int nMaxLen = -1;
CString strLine;
int nSimDocCount = m_SimTable.GetDocCount();
for (int i=0; i<nSimDocCount; i++) {
    // get next document entry from the similarity table
    CSimEntry* pSimEntry = m_SimTable.GetSimEntry(i);
    int nDocID = pSimEntry->nDocID;
    strLine.Format(_T("[%d]-[%d]:"), m_iCurrDoc, nDocID);
    m_fileSimMatrix.WriteString(strLine);

    // get the number of single word matches
    int nSingleCount = 0;
    CSingleMatchEntry* pSingleMatch = m_SimTable.GetSingleMatchEntry(nDocID);
    if (pSingleMatch)
        nSingleCount = pSingleMatch->GetMatchCount();
    strLine.Format(_T("(single: %d)"), nSingleCount);
    m_fileSimMatrix.WriteString(strLine);

    // get the number of phrase matches
    int nSntCount = pSimEntry->GetSentenceCount();
    strLine.Format(_T("(phrases: %d)"), nSntCount);
    m_fileSimMatrix.WriteString(strLine);

    // build a histogram for matching phrase lengths
    nMaxLen = -1;
    ZeroMemory(histogram, sizeof(int)*MAX_MATCH_LENGTH_BINS);
    CSimSentence* pSimSnt = NULL;
    for (int j=0; j<nSntCount; j++) {
        pSimSnt = pSimEntry->GetSimSentence(j);
        histogram[pSimSnt->nLength]++;
        if (pSimSnt->nLength > nMaxLen) nMaxLen = pSimSnt->nLength;
    }

    // output histogram
    for (j=0; j<=nMaxLen; j++) {
        if (histogram[j]) {
            strLine.Format(_T("(%d,x%d)"), j, histogram[j]);
            m_fileSimMatrix.WriteString(strLine);
        }
    }
    m_fileSimMatrix.WriteString(_T("\n"));
}
}

bool CDSCEngine::Initialize(void)
{
    ResetState();
if (m_ClusteringMethod == CLUSTERING_METHOD::AHC) {
    theApp.m_AhcClusterMgr.SetMaxClusters(m_nNumClusters);
}
return true;
}

bool CDSCEngine::Finalize(void)
{
    if (m_fileSimMatrix.m_pStream)
        m_fileSimMatrix.Close();

    CDSCApp* pApp = (CDSCApp*)AfxGetApp();

    if (m_ClusteringMethod == CLUSTERING_METHOD::AHC)
        ((AhcClusterMgr*)m_pClusterMgr)->ProcessData();

    m_pClusterMgr->OutputClusters("clusters.txt");

    CSdioFile simFile;
    if (!simFile.Open("similarity.txt", CFile::modeCreate | CFile::modeWrite | CFile::typeText))
        AfxMessageBox("Cannot create, similarity, output, file.");
    else
        theApp.m_SimilarityMatrix.Output(simFile);

    return true;
}

void CDSCEngine::CalcDocSim(CDocSim* pDocSim)
{
    CString strLine;

    int nSimDocCount = m_SimTable.GetDocCount();

    if (nSimDocCount == 0) {
        theApp.m_SimilarityMatrix.AddNullSim(m_iCurrDoc);
        return;
    }

    for (int i = 0; i < nSimDocCount; i++) {
        // get next document entry from the similarity table
        CSimEntry* pSimEntry = m_SimTable.GetSimEntry(i);
        int nDocID = pSimEntry->nDocID;
        shelf[nDocID] = true; // mark this doc as seen
    }
double simPhrase = 0.0;
double simSingle = 0.0;
double dTemp = 0.0;

// process phrase matches
CSimSentence* pSimSnt = NULL;
int nSntCount = pSimEntry->GetSentenceCount();
for (int j=0; j<nSntCount; j++) { // for each matching phrase
    pSimSnt = pSimEntry->GetSimSentence(j);
    ASSERT(pSimSnt);
    if (m_Options.SkipAllStopWordsPhrases && pSimSnt->IsStopWords()) // skip sentences that are all-stop-words
        continue;
    dTemp = pSimSnt->nLength * (pSimSnt->nCountX*pSimSnt->fWeightX + pSimSnt->nCountY*pSimSnt->fWeightY); // length * (w1*c1 + w2*c2)
    dTemp *= dTemp; // square it
    simPhrase += dTemp; // summation
}
if (nSntCount != 0) {
    simPhrase = sqrt(simPhrase); // square root of summation
    simPhrase /= (m_DocNorm[m_iCurrDoc] + m_DocNorm[nDocID]);
    simPhrase = pow(simPhrase, 0.5); // experimenting with expanding the sim value at low ends
}
if (simPhrase > 1.0) simPhrase = 1.0;

// process single word matches
CSingleMatchEntry* pSingleMatch = pSimEntry->pSingleMatch;
if (pSingleMatch) {
    int nSingleCount = pSingleMatch->GetMatchCount();
    CGraphNode* pNode = NULL;
    DSC_TERM_FREQ* pTFreq1, * pTFreq2;
    double dW1 = 0.0, dW2 = 0.0, dCos = 0.0;
    for (j=0; j<nSingleCount; j++) {
        pNode = pSingleMatch->GetNode(j);
        pTFreq1 = pNode->GetTermFreq(m_iCurrDoc);
        pTFreq2 = pNode->GetTermFreq(nDocID);
        dW1 = pTFreq1->nVeryHigh*SIG_VERY_HIGH + pTFreq1->nHigh*SIG_HIGH
             + pTFreq1->nMedium*SIG_MEDIUM + pTFreq1->nLow*SIG_LOW;
        dW2 = pTFreq2->nVeryHigh*SIG_VERY_HIGH + pTFreq2->nHigh*SIG_HIGH
             + pTFreq2->nMedium*SIG_MEDIUM + pTFreq2->nLow*SIG_LOW;
        dCos += dW1*dW2;
    }
    double denom = m_DocSingleNorm[m_iCurrDoc] * m_DocSingleNorm[nDocID];
    denom = sqrt(denom);
```cpp
simSingle = dCos/denom;
// experimenting with expanding the sim value at low ends
if (simSingle > 1.0) simSingle = 1.0;

// send two sim values to SimilarityMatrix
theApp.m_SimilarityMatrix.AddSim(m_iCurrDoc, nDocID, simSingle, simPhrase);

// blend two sim's
double simTotal = 0.0;
#define DEFAULT_BLEND_FACTOR 0.8

double alpha = m_dSimBlendFactor;
simTotal = alpha*simPhrase + (1-alpha)*simSingle;
pDocSim->AddSim(nDocID, simTotal);
strLine.Format("%d-%d:0.5%fsingle(%f)_phrase(%f)\n", m_iCurrDoc, nDocID, simTotal, simSingle, simPhrase);
m_fileSimMatrix.WriteString(strLine);
// get the number of single word matches
m_fileSimMatrix.WriteString("----\n\n");
}

void CDSCENgine::SetSimBlendFactor(double dBlendFactor)
{
   m_dSimBlendFactor = dBlendFactor;
}
```

```
// WordGraph.h: interface for the CWordGraph class.
//
// ************************************************************************
#endif
#endif
```

```
#if !defined(AFX_WORDGRAPH_H__46591113_17DF_4DDE_A706_812F95FA5124_INCLUDED_)
define AFX_WORDGRAPH_H__46591113_17DF_4DDE_A706_812F95FA5124_INCLUDED_

// Added by ClassView
// #include "strtree.h" // don't need it right now
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
```
#include "atlcoll.h"

class CGraphNode;
class CGraphEdge;

class CWordGraph : public CObject
{
public:
  CWordGraph();
  virtual ~CWordGraph();

cGraphNode* AddNode(CString strWord);
GraphNode* FindNode(CString strWord);
int LinkNodes(CGraphNode* pSrc, CGraphNode* pDest);
int GetNodeCount(void) { return m_mapIndex.GetCount(); }
int GetEdgeCount(void) { return m_nEdgeCount; }
void Clear(void);

protected:
  CGraphNode* AddNodeToMap(CGraphNode* pNode);
  CGraphNode* AddNodeToTree(CGraphNode* pNode);

  // implementation of the graph nodes index (Hashing Table)
  CAtlMap< CString, CGraphNode*, CStringElementTraits<CString> > m_mapIndex;

private:
  CGraphNode* m_pPreAllocatedNode;
  int m_nEdgeCount;
};

#endif // !defined(AFX_WORDGRAPH_H__46591113_17DF_4DDE_A706_812F95FA5124__INCLUDED_)
```c
#define _DEBUG
#define THIS_FILE
static char THIS_FILE[]=__FILE__;
#define new DEBUG_NEW
#endif

#define INITIAL_MAP_SIZE 10007

CWordGraph::CWordGraph()
: m_nEdgeCount(0)
{
    // Setup a pre-allocated graph node for new node additions
    m_pPreAllocatedNode = new CGraphNode;
    m_mapIndex.InitHashTable(INITIAL_MAP_SIZE);
}

CWordGraph::~CWordGraph()
{
    if (m_pPreAllocatedNode)
        delete m_pPreAllocatedNode;
    Clear();
}

// Adds a new node to the graph and returns a pointer to it
// if the node already exists, it returns the existing node pointer
CGraphNode* CWordGraph::AddNode(CString strWord)
{
    CGraphNode* pNode = m_pPreAllocatedNode;
    pNode->SetWord(strWord);
    CGraphNode* pTempNode = AddNodeToMap(pNode);
    if (pTempNode == pNode)
        // we just added a new word, create a new pre-allocated node before
        // returning the original's pointer
        m_pPreAllocatedNode = new CGraphNode;
    return pTempNode;
}

CGraphNode* CWordGraph::AddNodeToMap(CGraphNode* pNode)
{
```
CGraphNode* pTempNode = NULL;

// if word is already in hash table, return the corresponding node pointer
if (m_mapIndex.Lookup(pNode->GetWord(), pTempNode))
    return pTempNode;

// word not found, add it to hash table and return the passed pointer
m_mapIndex[pNode->GetWord()] = pNode;
return pNode;
}

int CWordGraph::LinkNodes(CGraphNode* pSrc, CGraphNode* pDest)
{
    m_nEdgeCount++;
    return pSrc->LinkTo(pDest);
}

void CWordGraph::Clear(void)
{
    // clean up allocated nodes by iterating through the hash table
    POSITION pos = m_mapIndex.GetStartPosition();
    CString strKey;
    CGraphNode* pNode = NULL;
    while (pos != NULL) {
        m_mapIndex.GetNextAssoc(pos, strKey, pNode);
        if (pNode)
            delete pNode;
    }
    m_mapIndex.RemoveAll();
    m_nEdgeCount = 0;
}

// SimTable.h: interface for the CSimTable class.
//
#endif

#if !defined(AFX_SIMTABLE_H__EA7CCBD7_C8C9_44E4_9441_43A92CD154F4__INCLUDED__)
define AFX_SIMTABLE_H__EA7CCBD7_C8C9_44E4_9441_43A92CD154F4__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include "GraphNode.h"
#include "DocInfo.h"
#include "afxtempl.h"

class CSimSentence : public CObject
{
public:
    int nSID;
    int nLength;
    int nSeq;
    DSC_SIGLEVEL sigLevelX; // owner doc level
    DSC_SIGLEVEL sigLevelY; // other doc level
    int nCountX; // several entries could be merged if they represent the same phrase
        // nCountX represents how many times this phrase occurred in the "owner document"
    int nCountY; // nCountY represents how many times this phrase occurred in the "other document"
    float fWeightX; // this is just the different sigLevels combined into one value
        // when this phrase is merged with other similar ones from the "owner document"
    float fWeightY; // combined sig level weights from "other document"

    // this CGraphNode* array keeps track of the actual nodes representing this sentence
    // so that we can retrieve the actual words of the sentence later
    CTypeCPtrArray<CObject, CGraphNode*> nodeArray;

    CSimSentence()
        : nSID(-1), nLength(0), nSeq(-1),
        sigLevelX(SIG_UNKNOWN), sigLevelY(SIG_UNKNOWN),
        nCountX(1), nCountY(1),
        fWeightX(0.0), fWeightY(0.0) {}
    CSimSentence(const DSC_SIMSENTENCE& snt) { SetData(snt); } 
    virtual ~CSimSentence() {} 

    void SetData(const DSC_SIMSENTENCE& snt)
    {
        nSID = snt.nSID;
        nSeq = snt.nSeq;
        nLength = snt.nLength;
        sigLevelX = snt.sigLevelX;
        sigLevelY = snt.sigLevelY;
        fWeightX = (float)sigLevelX;
        fWeightY = (float)sigLevelY;
        nCountX = 1;
nCountY = 1;
}

bool operator==(const CSimSentence& snt)
{  if (nSID != snt.nSID || nLength != snt.nLength || nSeq != snt.nSeq)
    return false;
  else return true;
}

bool IsWaining(const CSimSentence& snt)
{  int n1 = nodeArray.GetCount();
  int n2 = snt.nodeArray.GetCount();
  if (n1 != n2) return false;
  for (int i=0; i<n1; i++)
    if (nodeArray[i] != snt.nodeArray[i])
      return false;
  return true;
}

void GetSentence(CString& str)
{  int nCount = nodeArray.GetCount();
  if (nCount) str = nodeArray[0]->GetWord();
  for (int i=1; i<nCount; i++)
    str += _T('"');
  str += nodeArray[i]->GetWord();
}

bool IsStopWords()
{  CString str;
    GetSentence(str);
  int nCount = nodeArray.GetCount();
  for (int i=0; i<nCount; i++)
    if (!nodeArray[i]->IsStopWord()) break;
  return (i >= nCount);
};

class CSingleMatchEntry;

class CSimEntry : public CObject
{
public:
  int nDocID;
CTypedPtrArray<CBArray, CSimSentence*> sntTable;
CSingleMatchEntry* pSingleMatch;

CSimEntry(): pSingleMatch(NULL) {}
virtual ~CSimEntry() { Empty(); }

// Find the sentence entry with the given sentence ID AND word sequence number
CSimSentence* FindSentence(DSC_SIMSENTENCE& sntSim)
{
    int nSize = sntTable.GetSize();
    for (int i=0; i<nSize; i++)
        if (sntTable[i]->nSID == sntSim.nSID) // is this the sentence we are looking for?
            if (sntTable[i]->nSeq == (sntSim.nSeq)) // if so, is the given sequence number the same as the stored seq. no.?
                return sntTable[i]; // sentence found, return its pointer
    return NULL;
}

void Update(DSC_SIMSENTENCE& sntSim, CGraphNode* pPrevNode, CGraphNode* pNode)
{
    CSimSentence* pSnt = FindSentence(sntSim); // locate sentence entry in similar sentences list
    if (pSnt) { // if found, extend sentence match
        pSnt->nSeq++;
        pSnt->nLength++;
        pSnt->nodeArray.Add(pNode); // record the new matching node
    }
    else if (0) // no sentence match found, add a new sentence match
        sntSim.nLength = 2; // a new sentence match is always a 2-word length match
        sntSim.nSeq++; // match upto the following word, not the current
    CSimSentence* pSnt = new CSimSentence(sntSim);
    pSnt->nodeArray.Add(pPrevNode); // record the two found matching nodes
    pSnt->nodeArray.Add(pNode);
    sntTable.Add(pSnt);
}

int GetSentenceCount() { return sntTable.GetCount(); }

CSimSentence* GetSimSentence(int index)
{ ASSERT(index>=0 && index<=sntTable.GetCount());
return sntTable[index];
}

void Empty()
{
    for (int i=0; i<sntTable.GetCount(); i++)
        if (sntTable[i])
            delete sntTable[i];
    sntTable.RemoveAll();
}

// this will combine multiple matches from the owner document
// to one match from the other document
// (combines siglevels of owner matches and adjusts owner match count)
void MergeSentenceX(CSimSentence* pSnt)
{
    int nMatches = 0;
    float fWeight = 0;
    for (int i=0; i<sntTable.GetCount(); i++)
        if (sntTable[i] == pSnt) continue; // skip oneself
        if (*pSnt == *(sntTable[i])) { // if we got a match
            nMatches++;
            fWeight += (float)(sntTable[i]->sigLevelX); // accumulate
            sigLevels (weights)
            delete sntTable[i]; // discard sentence as we no longer need it
            sntTable.RemoveAt(i);
    }
    if (nMatches == 0) // if we there were no duplicates, return
        return;
    nMatches++; // add oneself
    fWeight += (float)(pSnt->sigLevelX); // add our weight
    fWeight /= nMatches;
    pSnt->nCountX = nMatches; // adjust owner match count
    pSnt->fWeightX = fWeight; // adjust owner combined weight
}

// this will combine multiple matches from the other document
// to one match from the owner document
// (combines siglevels of other doc matches and adjusts other doc match count)
void MergeSentenceY(CSimSentence* pSnt)
{
    int nMatches = 0;
    float fWeight = 0;
    for (int i=0; i<sntTable.GetCount(); i++)
        if (sntTable[i] == pSnt) continue; // skip oneself
        if (pSnt->IsMatching(*(sntTable[i]))) { // if we got a match
            nMatches++;
```cpp
fWeight += (float)(sntTable[i]->sigLevelY); // accumulate
sigLevels (weights)
delete sntTable[i]; // discard sentence as we no longer need it
sntTable.RemoveAt(i);
}
if (nMatches == 0) // if we there were no duplicates, return
return;
nMatches++; // add oneself
fWeight += (float)(pSnt->sigLevelY); // add this phrase weight
fWeight /= nMatches;
pSnt->nCountY = nMatches; // adjust other doc match count
pSnt->fWeightY = fWeight; // adjust other doc combined weight
}

void Merge()
{
    // first merge owner doc multiple matches
    for (int i=0; i<sntTable.GetCount(); i++)
        MergeSentenceX(sntTable[i]);
    // then merge other doc multiple matches
    for (int i=0; i<sntTable.GetCount(); i++)
        MergeSentenceY(sntTable[i]);
}

class CSingleMatchEntry : public CObject
{
    public:
        int nDocID;
        CTypedPtrArray<CObject, CGraphNode> nodeTable;

    CSingleMatchEntry(int doc = -1): nDocID(doc) {}  
    ~CSingleMatchEntry() { nodeTable.RemoveAll(); }  
    bool FindNode(CGraphNode* pNode)
    {
        for (int i=0; i<nodeTable.GetCount(); i++)
            if (nodeTable[i] == pNode)
                return true;
        return false;
    }
    bool AddNode(CGraphNode* pNode)
    {
        if (!FindNode(pNode))
            {
                nodeTable.Add(pNode);
                return true;
            }
        return false;
    }
};
```
}  
CGraphNode* GetNode(int index)  
{  
    ASSERT(index>=0);  
    if (index<nodeTable.GetCount())  
        return nodeTable[index];  
    else  
        return NULL;  
}  
int GetMatchCount() { return nodeTable.GetCount(); }  
};

class CSimTable : public CObject  
{
    public:  
        void Update(int nDocID, DSC_SIMSENTENCE* sntSim, CGraphNode* pPrevNode,  
                        CGraphNode* pNode);  
        CSimEntry* FindDoc(int nDocID);  
    void SetDocID(int nDocID) { m_nDocID = nDocID; }  
        CSimTable(int nDocID = -1);  
    virtual ~CSimTable();  
    int GetDocCount() { return m_SimTable.GetCount(); }  
    CSimEntry* GetSimEntry(int index)  
    {  
        ASSERT(index>=0 && index<=m_SimTable.GetCount());  
        return m_SimTable[index];  
    }  
    void MergeDuplicates() // merge duplicate phrase matches in each document  
                entry  
    {  
        int nCount = GetDocCount();  
        for (int i=0; i<nCount; i++) {  
            m_SimTable[i]->Merge();  
        }  
    }  
// Removes all entries  
void RemoveAll(void);

    CSingleMatchEntry* FindSingleMatch(int nDocID)  
    {  
        for (int i=0; i<m_SingleMatchTable.GetCount(); i++)  
            if (m_SingleMatchTable[i]->nDocID == nDocID)  
                return m_SingleMatchTable[i];  
        return NULL;  
    }
void AddSingleMatch(CGSession* pNode)
{
    CSingleMatchEntry* pSingleMatch = NULL;
    int nCount = pNode->GetDocCount();
    for (int i=0; i<nCount; i++) {
        int nDocID = pNode->GetDocID(i);
        if (nDocID != m_nDocID) { /* don't add our own document */
            pSingleMatch = FindSingleMatch(nDocID);
            if (pSingleMatch) {
                pSingleMatch = new CSingleMatchEntry(nDocID);
                m_SingleMatchTable.Add(pSingleMatch);
                // add a link to the single match entry in the phrase sim entry with the same DocID
                CSimEntry* pSimEntry = FindDoc(nDocID);
                if (pSimEntry) { /* if not, add new entry */
                    pSimEntry = new CSimEntry;
                    pSimEntry->nDocID = nDocID;
                    m_SimTable.Add(pSimEntry);
                }
                pSimEntry->pSingleMatch = pSingleMatch;
            }
        }
        pSingleMatch->AddNode(pNode);
    }
}

int GetSingleMatchCount() { return m_SingleMatchTable.GetCount(); }

CGSession* GetSingleMatch(int nDocID, int index)
{ CSingleMatchEntry* pMatch = FindSingleMatch(nDocID);
    if (pMatch)
        return NULL;
    return pMatch->GetNode(index);
}

CSingleMatchEntry* GetSingleMatchEntry(int nDocID)
{ return FindSingleMatch(nDocID); }

protected:
    int m_nDocID;
    CTypedPtrArray<CGSession, CSimEntry*> m_SimTable;
    CTypedPtrArray<CGSession, CSingleMatchEntry*> m_SingleMatchTable;
};

#endif // !defined(AFX_SIMTABLE_H__EA7C0D7C8C9_44E4_9441_43A92CD154F4__INCLUDED_)
// SimTable.cpp: implementation of the CSimTable class.
//
Thông tin này có thể được dịch tự động.
#include "stdafx.h"
#include "DSC.h"
#include "SimTable.h"

#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[]=__FILE__;
#define new DEBUG_NEW
#endif

// Construction/Destruction

CSimTable::CSimTable(int nDocID /*=-1*/) : m_nDocID(nDocID)
{
}

CSimTable::~CSimTable()
{
    // clean up similarity table
    RemoveAll();
}

void CSimTable::Update(int nDocID, DSC_SIMSENTEENCE &sntSim, CGraphNode* pPrevNode, CGraphNode* pNode)
{
    // locate document entry
    CSimEntry* pEntry = FindDoc(nDocID);
    if (!pEntry) { // if not, add new entry
        pEntry = new CSimEntry;
        pEntry->nDocID = nDocID;
        m_SimTable.Add(pEntry);
    }
    pEntry->Update(sntSim, pPrevNode, pNode); // update entry
}

CSimEntry* CSimTable::FindDoc(int nDocID)
{
    int i=0;
}
Implementation

```c++
while (i < m_SimTable.GetSize()) {
    if (m_SimTable[i]->nDocID == nDocID)
        return m_SimTable[i];
    i++;
}
return NULL;
}

// Removes all entries
void CSimTable::RemoveAll(void)
{
    // remove all sentence similarity entries from the table
    int nSize = m_SimTable.GetSize();
    for (int i=0; i<nSize; i++)
        if (m_SimTable[i])
            delete m_SimTable[i];
    m_SimTable.RemoveAll();

    // remove all single term entries
    nSize = m_SingleMatchTable.GetSize();
    for (int i=0; i<nSize; i++)
        if (m_SingleMatchTable[i])
            delete m_SingleMatchTable[i];
    m_SingleMatchTable.RemoveAll();
}

// ClusterMgr.h: interface for the ClusterMgr class.
//
// ///////////////////////////////////////////////////////////////////////////
#pragma once

#include <atlcoll.h>

// dependencies
class Cluster;
class CDocSim;

class ClusterMgr
{
protected:
    enum ClusterType {
        DSC,
        AHC
    };
    // attributes
```
CAtlList<Cluster*> m_ClusterList; // list of clusters

// protected methods
double Fmeasure(int nClassID); // calculate F-measure for class(i)

double m_dFmeasure;
double m_dEntropy;
double m_dOverallSim;

public:
ClusterMgr(void);
virtual ~ClusterMgr(void);

virtual Cluster* CreateCluster(CRuntimeClass* pRTClass);
virtual CDocSim* GetDocSim(int nDocID);

virtual void AddDoc(CDocSim* pDocSim) {}
virtual void ProcessDocument(int nDocID) {};

virtual void RemoveAll(void);
virtual void OutputClusters(const CString& strFilename);

virtual double Fmeasure(void);
virtual double Entropy(void);
virtual double OverallSimilarity(void);

virtual int GetClusterCount(void) { return m_ClusterList.GetCount(); };
virtual double GetFmeasure(void)
{ if (m_dFmeasure == -1)
    return Fmeasure();
  else
    return m_dFmeasure;
}
virtual double GetEntropy(void)
{ if (m_dEntropy == -1)
    return Entropy();
  else
    return m_dEntropy;
}
virtual double GetOverallSimilarity(void)
{ if (m_dOverallSim == -1)
    return OverallSimilarity();
  else
    return m_dOverallSim; }
Implementation

};

// ClusterMgr.cpp: implementation of the ClusterMgr class.
//
// //////////////////////////////////////////////////////////////////////////
#include "stdafx.h"
#include "clusermgr.h"

// dependencies
#include "Cluster.h"
#include "DSC.h"
#include "DocSim.h"
#include <vector>
#include <algorithm>

extern CDSCApp theApp;

ClusterMgr::ClusterMgr(void)
 : m_dFmeasure(-1)
 , m_dEntropy(-1)
 , m_dOverallSim(-1)
{
}

ClusterMgr::~ClusterMgr(void)
{
   RemoveAll();
}

// create a new cluster object and add it to the cluster list
Cluster* ClusterMgr::CreateCluster(CRuntimeClass* pRTClass)
{
   if (!pRTClass)
      return NULL;

   int nID = m_ClusterList.GetCount();
   // Cluster* pCluster = new Cluster(nID, this);

   Cluster* pCluster = (Cluster*)pRTClass->CreateObject();
pCluster->SetID(nID);
pCluster->SetManager(this);

   m_ClusterList.AddTail(pCluster);
   return pCluster;
void ClusterMgr::RemoveAll(void)
{
    // clean up cluster list
    if (!m_ClusterList.IsEmpty()) {
        POSITION pos = m_ClusterList.GetHeadPosition();
        while (pos)
            delete m_ClusterList.GetNext(pos);
    }
    m_ClusterList.RemoveAll();

    m_dFmeasure = -1;
    m_dEntropy = -1;
    m_dOverallSim = -1;
}

CDocSim* ClusterMgr::GetDocSim(int nDocID)
{
    return theApp.m_SimilarityMatrix.GetDocSim(nDocID);
}

void ClusterMgr::OutputClusters(const CString& strFilename)
{
    CStdioFile outfile;
    if (!outfile.Open(strFilename, CFile::modeCreate | CFile::modeWrite | CFile::typeText))
        return;

    if (!m_ClusterList.IsEmpty()) {
        // output F-measure
        CString strLine;
        if (m_dFmeasure == -1)
            m_dFmeasure = Fmeasure();
        strLine.Format("F-measure = %f\n", m_dFmeasure);
        outfile.WriteString(strLine);

        if (m_dEntropy == -1)
            m_dEntropy = Entropy();
        strLine.Format("Entropy = %f\n", m_dEntropy);
        outfile.WriteString(strLine);

        if (m_dOverallSim == -1)
            m_dOverallSim = OverallSimilarity();
        strLine.Format("Overall Similarity = %f\n", m_dOverallSim);
    }
outfile.WriteString(strLine);

    // output individual clusters
    Cluster* pCluster = NULL;
    POSITION pos = m_ClusterList.GetHeadPosition();
    while (pos) {
        pCluster = m_ClusterList.GetNext(pos);
        pCluster->Output(outfile);
    }
    }
    }

    // F-measure of class (i)
    // F(i) = max(F(i,j))
    // where:
    // F(i,j) : F-measure of class (i) w.r.t. cluster (j)
    double ClusterMgr::Fmeasure(int nClassID)
    {
        double F = 0.0;
        double Fmax = 0.0;

        Cluster* pCluster = NULL;
        Cluster* pCMax = NULL;

        POSITION pos = m_ClusterList.GetHeadPosition();
        while (pos) {
            pCluster = m_ClusterList.GetNext(pos);
            F = pCluster->Fmeasure(nClassID);
            if (F > Fmax) {
                Fmax = F;
                pCMax = pCluster;
            }
        }

        pCMax->SetClassMapping(nClassID);
        return Fmax;
    }

double ClusterMgr::Fmeasure(void)
{
    int nClassDocCount = 0;
    int nTotalDocCount = 0;
    int nClassCount = theApp.m_DocClassMgr.GetClassCount();
    double F = 0.0;
    for (int i=0; i<nClassCount; i++) { // for each class
nClassDocCount = theApp.m_DocClassMgr.GetClassDocCount(i);
nTotalDocCount += nClassDocCount;
F += nClassDocCount * Fmeasure(i);
}
F /= nTotalDocCount;

m_dFmeasure = F;
return F;
}

double ClusterMgr::Entropy(void)
{
    double E = 0.0;
    double p = 0.0;
    Cluster* pCluster = NULL;
    int nTotalDocCount = 0;

    POSITION pos = m_ClusterList.GetHeadPosition();
    while (pos) {
        pCluster = m_ClusterList.GetNext(pos);
        nTotalDocCount += pCluster->GetDocCount();
    }

    pos = m_ClusterList.GetHeadPosition();
    while (pos) {
        pCluster = m_ClusterList.GetNext(pos);
        p = pCluster->GetDocCount()/(double)nTotalDocCount;
        E += p * pCluster->Entropy();
    }

    m_dEntropy = E;
    return E;
}

double ClusterMgr::OverallSimilarity(void)
{
    double S = 0.0;
    double p = 0.0;
    Cluster* pCluster = NULL;
    int nTotalDocCount = theApp.m_DocClassMgr.GetTotalDocCount();

    POSITION pos = m_ClusterList.GetHeadPosition();
    while (pos) {
        pCluster = m_ClusterList.GetNext(pos);
        p = pCluster->GetDocCount()/(double)nTotalDocCount;

Implementation

```cpp
S += p * pCluster->OverallSimilarity();
}
m_dOverallSim = S;
return S;
}

// Cluster.h: interface for the Cluster class.
//
// //////////////////////////////////////////////////////////////////////////

#pragma once

#include "atlcoll.h"

class ClusterMgr;
class CDocSim;

struct DocEntry {
    CDocSim* pDocSim;
    double dMembership;
    double dHrOut; // Histogram Ratio if the document is removed from the cluster
};

class Cluster : public CObject
{
public:
    // declare support for dynamic object creation
    DECLARE_DYNCREATE(Cluster)

protected:
    // attributes
    int m_nID;
    ClusterMgr* m_pMgr;
    CAtlMap<int, DocEntry> m_DocTable;
    double m_dFmeasure;
    double m_dEntropy;
    double m_dOverallSim;
    int m_nMappedClass;

public:
    // construction/destruction
    Cluster(void) // default constructor for dynamic object creation
{ Cluster(-1, NULL); }
Cluster(int nID, ClusterMgr* pMgr);
virtual ~Cluster(void);

virtual void AddDoc(int nDocID);
virtual void SetManager(ClusterMgr* pMgr) { m_pMgr = pMgr; }
virtual void SetID(int nID) { m_nID = nID; }

virtual int GetDocCount(void)
{ return m_DocTable.GetCount(); }
virtual void SetClassMapping(int nClassID)
{ m_nMappedClass = nClassID; }
virtual int GetClassOverlap(int nClassID);

virtual double Precision(int nClassID);
virtual double Recall(int nClassID);
virtual double Fmeasure(int nClassID);
virtual double Entropy(void);
virtual double OverallSimilarity(void);

virtual void Output(CStdioFile& outfile);
};

// Cluster.cpp: implementation of the Cluster class.
//
#pragma once

#include "stdafx.h"
#include "cluster.h"
#include "math.h"

// dependencies
#include "DocSim.h"
#include "ClusterMgr.h"
#include "DocClassMgr.h"
#include "DSC.h"

extern CDSCApp theApp;

// support for dynamic object creation
IMPLEMENT_DYNCREATE(Cluster, CObject)
Implementation

```
Cluster::Cluster(int nID, ClusterMgr* pMgr)
    : m_pMgr(pMgr)
    , m_nID(nID)
    , m_nMappedClass(-1)
    , m_dMeasure(0)
    , m_dEntropy(0)
    , m_dOverallSim(0)
{
}

Cluster::~Cluster()
{
}

void Cluster::AddDoc(int nDocID)
{
    CDocSim* pDocSim = theApp.m_SimilarityMatrix.GetDocSim(nDocID);

    DocEntry docEntry;
    docEntry.dMembership = 1.0;
    docEntry.pDocSim = pDocSim;
    m_DocTable[nDocID] = docEntry;
}

int Cluster::GetClassOverlap(int nClassID)
{
    int nCount = 0;
    int nDocID = 0;
    int nID = 0;
    POSITION pos = m_DocTable.GetStartPosition();
    while (pos) {
        nDocID = m_DocTable.GetNextKey(pos);
        nID = theApp.m_DocClassMgr.GetDocClass(nDocID);
        if (nID == nClassID)
            nCount++;
    }
    return nCount;
}

// Precision of cluster (j) w.r.t. class (i)
// P(i,j) = N(i,j)/N(i)
// where:
// N(i,j) : # of members of class (i) in cluster (j)
// N(i)   : # of members of class (i)
double Cluster::Precision(int nClassID)
```
{
    int Nij = GetClassOverlap(nClassID);
    int Ni = theApp.m_DocClassMgr.GetClassDocCount(nClassID);
    return (double)Nij/Ni;
}

// Recall of cluster (j) w.r.t. class (i)
// R(i,j) = N(i,j)/N(j)
// where:
// N(i,j) : # of members of class (i) in cluster (j)
// N(j) : # of members of cluster (j)
double Cluster::Recall(int nClassID)
{
    int Nij = GetClassOverlap(nClassID);
    int Nj = m_DocTable.GetCount();
    return (double)Nij/Nj;
}

// F-measure of cluster (j) w.r.t. class (i)
// F = 2*P(i,j)*R(i,j) / (P(i,j)+R(i,j))
double Cluster::Fmeasure(int nClassID)
{
    double P = Precision(nClassID);
    double R = Recall(nClassID);
    if (P==0 && R==0)
        return 0;
    else
        return 2*P*R/(P+R);
}

// Entropy of cluster
// E = - 1/log(c) * sum( p(i)*log(p(i)) )
// where:
// sum is taken over all classes
// c : # of classes
// p(i) : fraction of documents from class (i) in this cluster
double Cluster::Entropy(void)
{
    double E = 0.0;
    int nClassID = 0;
    int nDocID = 0;
    int nClassCount = theApp.m_DocClassMgr.GetClassCount();
    int* aClassOverlap = new int[nClassCount];
    ZeroMemory(aClassOverlap, sizeof(int)*nClassCount);
IMPLEMENTATION

POSITION pos = m_DocTable.GetStartPosition();
while (pos) {
    nDocID = m_DocTable.GetNextKey(pos);
    nClassID = theApp.m_DocClassMgr.GetDocClass(nDocID);
    aClassOverlap[nClassID]++;
}

double p = 0;
for (int i=0; i<nClassCount; i++) {
    p = (double)aClassOverlap[i]/theApp.m_DocClassMgr.GetClassDocCount(i);
    if (p != 0)
        E += p*log(p);
}
if (nClassCount == 1)
    E = -E;
else
    E = - (1.0/log(nClassCount)) * E;

m_dEntropy = E;

delete [] aClassOverlap;
return E;
}

// Overall similarity of cluster
// S = 1/(N-2) * sum sim(di,dj)
// where:
// N: # of documents in this cluster
// sum: summation of all similarities in the cluster
// between every document and the other
// sim(): similarity
// di,dj: documents belonging to this cluster
double Cluster::OverallSimilarity(void)
{
    double dSim = 0.0;
    int nTotalCount = 0;
    int nDocID = 0;
    CDocSim* pDocSim = NULL;
    POSITION pos = m_DocTable.GetStartPosition();
    while (pos) {
        nDocID = m_DocTable.GetNextKey(pos);
        //pDocSim = m_DocTable.GetNextValue(pos);
        pDocSim = m_pMgr->GetDocSim(nDocID);

        int nDocID2 = 0;
POSITION pos2 = pDocSim->m_SimTable.GetStartPosition();
while (pos2) {
    CAtlMap<unsigned long, double>::CPair* pPair = pDocSim->m_SimTable.
            GetNext(pos2);
    nDocID2 = pPair->m_key;
    // nDocID2 = pDocSim->m_SimTable.GetKeyAt(pos2);
    if (m_DocTable.Lookup(nDocID2)) {  
        // similarities are doubled to reflect doc1 -> doc2 similarity  
        // and doc2 -> doc1 similarity (same value, but two directions)  
        dSim += 2 * pDocSim->m_SimTable.GetNextValue(pos2);
        dSim += 2 * pPair->m_value;
        nTotalCount += 2;
    }
}

// we even need to add self-similarity (equals 1.0) for each document
// this should be just the document count in the cluster times 1.0
dSim += m_DocTable.GetCount();

nTotalCount += m_DocTable.GetCount();
dSim /= nTotalCount;

m_dOverallSim = dSim;
return dSim;
}

void Cluster::Output(CStdioFile& outfile)
{
    CString strLine;
    strLine.Format("Cluster[%d], tMapped, tClass, tSimilarity, tOverall

    outfile.WriteString(strLine);
    strLine.Format("Cluster, Entropy, Similarity, Overall Sim"

    outfile.WriteString(strLine);

    POSITION pos = m_DocTable.GetStartPosition();
    while (pos) {
        strLine.Format("%d", m_DocTable.GetNextKey(pos));
        outfile.WriteString(strLine);
    }
    outfile.WriteString("\n\n");
// DscClusterMgr.h: interface for the DscClusterMgr class.
//
#pragma once

#include <atlcoll.h>

// dependencies
#include "ClusterMgr.h"

class DscCluster;

class CDocSim;

#define DEFAULT_CLUSTERTHRESHOLD 0.275

class DscClusterMgr : public ClusterMgr
{
protected:
   // attributes
   double m_dClusterThreshold;

public:
   DscClusterMgr(void);
   ~DscClusterMgr(void);

   // methods
   DscCluster* CreateCluster();
   void AddDoc(CDocSim* pDocSim);
   void SetClusterThreshold(double dThreshold);
   double GetClusterThreshold(void) { return m_dClusterThreshold; }
   void ProcessDocument(int nDocID);

   void SetHrMin(double hr) { m_dHrMin = hr; }
   double GetHrMin() { return m_dHrMin; }
   void ReAssign(); // re-assign documents
   void SetReAssign(bool bRA) { m_bReAssign = bRA; }
   bool GetReAssign() { return m_bReAssign; }

protected:
   // minimum Histogram Ratio
   double m_dHrMin;
   bool m_bReAssign; // document re-assignment control flag
};

// DscClusterMgr.cpp: implementation of the DscClusterMgr class.
```c++
//
#include "stdafx.h"
#include "clustermgr.h"

// dependencies
#include "DocSim.h"
#include "DscCluster.h"
#include "DocClassMgr.h"
#include "DSC.h"
#include "dsscclustermgr.h"

extern CDSCApp theApp;

DscClusterMgr::DscClusterMgr(void)
    : m_dClusterThreshold(DEFAULT_CLUSTER_THRESHOLD), m_dIRMin(0), m_bReAssign(false)
    {
    }

// clean up
DscClusterMgr::~DscClusterMgr(void)
    {
        RemoveAll();
    }

DscCluster* DscClusterMgr::CreateCluster(void)
    {
        CRuntimeClass* pRTClass = RUNTIME_CLASS(DscCluster);
        return (DscCluster*)ClusterMgr::CreateCluster(pRTClass);
    }

void DscClusterMgr::AddDoc(CDocSim* pDocSim)
    {
        // copy this document's DocSim to our similarity matrix

        DscCluster* pCluster = NULL;
        bool bAdded = false;

        // try adding the document to each cluster
        POSITION pos = m_ClusterList.GetHeadPosition();
        while (pos)
        {
            pCluster = (DscCluster*)m_ClusterList.GetNext(pos);
            ASSERT(pCluster);
        }
    }
```
Implementation

```cpp
if (pCluster->AddDoc(pDocSim))
    bAdded = true;
}

if (m_bReAssign)
    ReAssign(); // reassign documents

if (!bAdded) { // if the document was not added to any cluster
    pCluster = CreateCluster(); // create a new cluster
    pCluster->AddFirstDoc(pDocSim); // add document to it
}
}

void DscClusterMgr::SetClusterThreshold(double dThreshold)
{
    if (dThreshold < 0) dThreshold = 0;
    if (dThreshold > 1) dThreshold = 1;
    m_dClusterThreshold = dThreshold;
}

void DscClusterMgr::ProcessDocument(int nDocID)
{
    CDocSim* pDocSim = GetDocSim(nDocID);
    AddDoc(pDocSim);
}

void DscClusterMgr::ReAssign(void)
{
    DscCluster* pCluster = NULL;
    POSITION pos = m_ClusterList.GetHeadPosition();
    int nBadDocID = -1;
    while (pos) {
        POSITION delPos = pos;
        pCluster = (DscCluster*)m_ClusterList.GetNext(pos);
        ASSERT(pCluster);
        CAtlArray<int> aBadDoc;
        pCluster->GetMaxOutDoc(aBadDoc);
        for (int i = 0; i < aBadDoc.GetCount(); i++) {
            int nBadDocID = aBadDoc[i];
            // try adding the bad document to other clusters
            DscCluster* pCluster2 = NULL;
            bool bAdded = false;
            POSITION pos2 = m_ClusterList.GetHeadPosition();
            while (pos2) {
                pCluster2 = (DscCluster*)m_ClusterList.GetNext(pos2);
            }
```
if (pCluster2 == pCluster) continue;
CDocSim* pDocSim = theApp.m_SimilarityMatrix.GetDocSim(nBadDocID);
if (pCluster2->AddDoc(pDocSim))
    bAdded = true;
}
if (bAdded) {
pCluster->RemoveDoc(nBadDocID);
if (pCluster->GetDocCount() == 0) {
    delete pCluster;
    pCluster = NULL;
    m_ClusterList.RemoveAt(delPos);
}
}
}
}

// DscCluster.h: interface for the DscCluster class.
//
#pragma once
#include <atlcoll.h>

// dependencies
#include "Cluster.h"

class CDocSim;
class DscClusterMgr;

// max number of cluster histogram bins
// bins are document similarity intervals
// each bin represents the number of similarities in that interval
// that appear in this cluster
#define MAX_SIMILARITY_BINS 10

class DscCluster : public Cluster
{
    DECLARE_DYNCREATE(DscCluster)

protected:
    // attributes
    unsigned long m_naHistogram[MAX_SIMILARITY_BINS];
public:
    // construction/destruction
    DscCluster(void) : Cluster(-1, NULL), m_dHR(0)
    {}      
    DscCluster(int nID, DscClusterMgr* pMgr);
    virtual ~DscCluster(void);

    // methods
    void AddFirstDoc(CDocSim* pDocSim);
    bool AddDoc(CDocSim* pDocSim, bool bTest = false);
    void UpdateHistogram(unsigned long naHistogram[], CDocSim* pDocSim, bool bRemove=false);
    double EvalHistogram(unsigned long naHistogram[]) const;
    void UpdateHrOut(CDocSim* pDocSimNew);

protected:
    // histogram ratio
    double m_dHR;

public:
    // returns the most bad document in the cluster (that is if removed will
cause maximum enhancement in the histogram ratio)
    void GetMaxOutDoc(CArray<int>& aBadDoc);
    void RemoveDoc(int nDocID);
};

#include "stdafx.h"
#include "dsccluster.h"

#include "DocSim.h"
#include "DscClusterMgr.h"

IMPLEMENT_DYNCREATE(DscCluster, Cluster)

DscCluster::DscCluster(int nID, DscClusterMgr* pMgr) : Cluster(nID, (ClusterMgr*)pMgr)
    , m_dHR(0)
{
    // initialize similarity histogram
    for (int i=0; i<MAX_SIMILARITY_BINS; i++)
    {  
        m_naHistogram[i] = 0;
    }  
}
DscCluster::~DscCluster(void)
{
    Cluster::~Cluster();
}

// Add the first document to cluster.
void DscCluster::AddFirstDoc(CDocSim* pDocSim)
{
    ASSERT(m_DocTable.IsEmpty()); // make sure there are no documents in cluster

    // initialize similarity histogram
    for (int i=0; i<MAX_SIMILARITY_BINS; i++)
        m_naHistogram[i] = 0;

    UpdateHistogram(m_naHistogram, pDocSim); // update cluster histogram
    DocEntry docEntry;
    docEntry.dMembership = 1.0;
    docEntry.pDocSim = pDocSim;
    docEntry.dHROut = 0.0;
    m_DocTable[pDocSim->GetDocID()] = docEntry; // add document to table
}

// Adds a document to the cluster.
// pDocSim: [in] pointer to document similarity table object
// bTest: [in] if true, add document in test mode, i.e. don't actually include it
// [retval] true: document enhances cluster quality so it is successfully added
// false: document did not enhance cluster quality and was not added
bool DscCluster::AddDoc(CDocSim* pDocSim, bool bTest)
{
    if (m_DocTable.Lookup(pDocSim->m_nDocID))
        return true;

    // prepare a copy of the actual histogram
    unsigned long tmpHistogram[MAX_SIMILARITY_BINS];
    for (int i=0; i<MAX_SIMILARITY_BINS; i++)
        tmpHistogram[i] = m_naHistogram[i];

    // update temp histogram with the new doc sim's
    UpdateHistogram(tmpHistogram, pDocSim);

    double qOld = EvalHistogram(m_naHistogram); // evaluate histogram BEFORE adding document
    double qNew = EvalHistogram(tmpHistogram); // evaluate histogram AFTER adding document

double dHRMin = ((DscClusterMgr*)m_pMgr)->GetHRMin();  // get the min
   histogram ratio

if (qNew >= dHRMin || (qNew > 0.0 && qNew >= qOld)) {  // if adding the
   document enhances cluster quality, add it to cluster
   if (!bTest) {  // if not in test mode, modify cluster histogram and
      document list
      UpdateHROut(pDocSim);  // update the out list
      CopyMemory(m_naHistogram, tmpHistogram, MAX_SIMILARITY_BINS*sizeof(
         unsigned long));
      DocEntry docEntry;
      docEntry.dMembership = 1.0;
      docEntry.pDocSim = pDocSim;
      docEntry.dHROut = qOld;
      m_DocTable[pDocSim->GetDocID()] = docEntry;  // add to document table
      of this cluster
      m_dHR = qNew;  // update histogram ratio
   }
   return true;  // inform caller that the document was added
}  
else  
   return false;  // otherwise inform caller that the document was not added

void DscCluster::UpdateHistogram(unsigned long naHistogram[], CDocSim* pDocSim,
   bool bRemove=*/false*/) {

   // iterate over the similarities in pDocSim
   // updating naHistogram with the new counts
   POSITION pos;
   unsigned long nDocID;
   double dValue;
   pos = pDocSim->m_SimTable.GetStartPosition();

   double dInterval = 1.0/MAX_SIMILARITY_BINS;  // histogram bin interval
   int iBin = 0;
   // iterate over similarities (from pDocSim to documents in the cluster)
   while (pos != NULL) {
      nDocID = pDocSim->m_SimTable.GetKeyAt(pos);
      if (m_DocTable.Lookup(nDocID)) {  // if document is in cluster,
         update histogram
         dValue = pDocSim->m_SimTable.GetNextValue(pos);
         iBin = (int)(dValue*MAX_SIMILARITY_BINS);  // determine bin
         number
if (iBin >= MAX_SIMILARITY_BINS) iBin = MAX_SIMILARITY_BINS-1; // make
sure we are within bounds
if (bRemove)
    naHistogram[iBin]--; // decrement count if removing document
else
    naHistogram[iBin]++; // increment count in that bin
}
else
dValue = pDocSim->m_SimTable.GetNextValue(pos);
}

// deal with backward similarities from documents in the cluster to this
document
pos = m_DocTable.GetStartPosition();
while (pos) {
    DocEntry docEntry = m_DocTable.GetNextValue(pos);
    CDocSim* pDocSim2 = docEntry.pDocSim;
    if (pDocSim2->m_SimTable.Lookup(pDocSim->m_nDocID)) {
        dValue = pDocSim2->m_SimTable[pDocSim2->m_nDocID];
        iBin = (int)(dValue*MAX_SIMILARITY_BINS); // determine bin
        number
        if (iBin >= MAX_SIMILARITY_BINS) iBin = MAX_SIMILARITY_BINS-1; // make
        sure we are within bounds
        if (bRemove)
            naHistogram[iBin]--; // decrement count if removing document
        else
            naHistogram[iBin]++; // increment count in that bin
    }
}

// Evaluate histogram by summing similarities count in the interval 0.5 - 1.0
// and dividing it by the total count of similarities.
// This should give us an idea of how coherent the cluster is.
double DscCluster::EvalHistogram(unsigned long naHistogram[]) const
{
    double dThreshold = DEFAULT_CLUSTER_THRESHOLD;
    if (m_pMgr)
        dThreshold = ((DscClusterMgr*)m_pMgr)->GetClusterThreshold();

    long lHighCount = 0;
    long lTotalCount = 0;
    double dInterval = 1.0/MAX_SIMILARITY_BINS; // histogram bin interval
Implementation

```c
int iBin = 0;
iBin = (int)(dThreshold*MAX_SIMILARITY_BINS); // determine bin number
if (iBin >= MAX_SIMILARITY_BINS) iBin = MAX_SIMILARITY_BINS-1; // make sure we are within bounds

for (int i=iBin; i<MAX_SIMILARITY_BINS; i++)
  lHighCount += naHistogram[i];

lTotalCount = lHighCount;
for (i=0; i<iBin; i++)
  lTotalCount += naHistogram[i];

if (lTotalCount == 0)
  return 0;

return (double)lHighCount/lTotalCount;
}

void DscCluster::UpdateHrOut(CDocSim* pDocSimNew)
{
  int nDocCount = GetDocCount() - 1; //exclude ourself
  if (nDocCount == 0) return;

  int nSimCount = nDocCount*(nDocCount+1)/2.0; // similarity count
  int iBin = (int)(((DscClusterMgr*)m_pMgr)->GetClusterThreshold()*
                   MAX_SIMILARITY_BINS);
  double T = (double)iBin/MAX_SIMILARITY_BINS;

  POSITION pos = pDocSimNew->m_SimTable.GetStartPosition();
  int nSimAboveT = 0;
  while (pos) {
    int nDocID = pDocSimNew->m_SimTable.GetKeyAt(pos);
    if (m_DocTable.Lookup(nDocID)) {
      if ((pDocSimNew->m_SimTable.GetValue(pos)>T))
        nSimAboveT++;
    } else
      pDocSimNew->m_SimTable.GetValue(pos);
  }

  pos = m_DocTable.GetStartPosition();
  DocEntry docEntry;
  double dHrNew = 0;
  while (pos) {
    int nDocID = m_DocTable.GetKeyAt(pos);
    docEntry = m_DocTable.GetValue(pos);
```
int nSimAboveNew = nSimAboveT;
if (pDocSimNew->m_SimTable[nDocID] > T)
    nSimAboveNew--;
dHRNew = (docEntry.dHROut * nSimCount + nSimAboveNew)/(double)(nSimCount +
nDocCount);
    docEntry.dHROut = dHRNew;
    m_DocTable[nDocID] = docEntry;
}

// returns the most bad document in the cluster (that is if removed will cause
// maximum enhancement in the histogram ratio)
void DscCluster::GetMaxOutDoc(CAtlArray<int>& aBadDoc)
{
    double dMaxHR = m_dHR;
    POSITION pos = m_DocTable.GetStartPosition();
    while (pos) {
        int nDocID = m_DocTable.GetKeyAt(pos);
        DocEntry docEntry = m_DocTable.GetNextValue(pos);
        if (docEntry.dHROut > m_dHR)
            aBadDoc.Add(nDocID);
    }
}

void DscCluster::RemoveDoc(int nDocID)
{
    CDocSim* pDocSim = m_DocTable[nDocID].pDocSim;
    UpdateHistogram(m_naHistogram, pDocSim, true);
    m_DocTable.RemoveKey(nDocID);
    m_dHR = EvalHistogram(m_naHistogram);
}
Bibliography


Index

Clustering
   Conceptual, 22
   Hierarchical, 17
   Partitional, 20

DC-tree Clustering, 56
Decision Trees, 22
Document Index Graph, 40
   construction, 45
   structure, 41

Incremental Clustering, 55

N-gram Representation, 12

Phrase Matching, 48
   algorithm, 48

Similarity Histogram, 57
Similarity Matrix, 15
Similarity Measure, 13
   blend factor, 53
   cosine correlation, 14
   Euclidean Distance, 14
   Hamming Distance, 14
   Jaccard, 14

Minkowski, 14
   phrase-based, 50
Tschebychev Distance, 14
Suffix Tree Clustering, 55

Term Frequency, 12

Vector Space Model, 12