

PREDICTIVE DATA MINING IN A COLLABORATIVE EDITING SYSTEM: THE  
*WIKIPEDIA* ARTICLES FOR DELETION PROCESS

by

ASHISH KUMAR ASHOK

B.Tech, Jawaharlal Nehru Technological University, 2009

A THESIS

submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences  
College of Engineering

KANSAS STATE UNIVERSITY  
Manhattan, Kansas

2011

Approved by:

Major Professor  
William H. Hsu

## Abstract

In this thesis, I examine the Articles for Deletion (AfD) system in *Wikipedia*, a large-scale collaborative editing project. Articles in *Wikipedia* can be nominated for deletion by registered users, who are expected to cite criteria for deletion from the *Wikipedia* deletion. For example, an article can be nominated for deletion if there are any copyright violations, vandalism, advertising or other spam without relevant content, advertising or other spam without relevant content. Articles whose subject matter does not meet the notability criteria or any other content not suitable for an encyclopedia are also subject to deletion.

The AfD page for an article is where *Wikipedians* (users of *Wikipedia*) discuss whether an article should be deleted. Articles listed are normally discussed for at least seven days, after which the deletion process proceeds based on community consensus. Then the page may be kept, merged or redirected, transwikied (i.e., copied to another Wikimedia project), renamed/moved to another title, userfied or migrated to a user subpage, or deleted per the deletion policy. Users can vote to keep, delete or merge the nominated article. These votes can be viewed in article's view AfD page. However, this polling does not necessarily determine the outcome of the AfD process; in fact, *Wikipedia* policy specifically stipulates that a vote tally alone should not be considered sufficient basis for a decision to delete or retain a page.

In this research, I apply machine learning methods to determine how the final outcome of an AfD process is affected by factors such as the difference between versions of an article, number of edits, and number of disjoint edits (according to some contiguity constraints). My goal is to predict the outcome of an AfD by analyzing the AfD page and editing history of the article. The technical objectives are to extract features from the AfD discussion and version history, as reflected in the edit history page, that reflect factors such as those discussed above, can be tested for relevance, and provide a basis for inductive generalization over past AfDs. Applications of such feature analysis include prediction and recommendation, with the performance goal of improving the precision and recall of AfD outcome prediction.

# Table of Contents

List of Figures .....	vi
Acknowledgements .....	vii
Chapter 1 - Introduction .....	1
1.1 Problem Definition .....	1
1.2 Goal and Technical Objectives .....	2
1.2.1 UNIX <code>Diff</code> command .....	4
1.2.2 Baseline0 .....	5
1.2.3 Baseline1 .....	5
1.2.4 Baseline2 .....	6
1.2.5 Baseline3 .....	6
1.2.6 <code>gSpan</code> Baseline .....	6
1.3 Overview .....	7
Chapter 2 - Background and Related work .....	8
2.1 Longest Common Subsequence Algorithm .....	8
2.2 Frequent pattern mining .....	10
2.2.1 <code>gSpan</code> (graph based substructure pattern mining) .....	10
2.3 Dotty (directed graph editor) .....	12
2.3 Counting the number of merge, keep and delete votes .....	13
2.4 Preparing the input .....	14
2.4.1 ARFF Format .....	14
2.5 Brief overview of <i>WEKA</i> .....	17
2.6 Machine Learning Algorithms Used For Classification .....	18
2.6.1 Classifier used in this project .....	18
Chapter 3 - Methodology .....	20
3.1 Extracting the AfD pages into the local system .....	20
3.2 Generating version graph from AfD articles .....	21
3.3 Input format to <code>gSpan</code> Algorithm .....	22
3.4 Input format for Arff data files .....	23
3.4.1 Baseline0 .....	23

3.4.2 Baseline1 .....	23
3.4.3 Baseline2 .....	24
3.4.4 Baseline3 .....	24
3.4.5 gSpan Baseline .....	24
Chapter 4 - Experimental Design.....	26
4.1 Sample text files given as initial input.....	26
4.2 Step by Step Generation of Version graph for sample text files: .....	26
Step1 .....	27
Step2 .....	27
Step3 .....	28
Step4 .....	29
Step5 .....	29
Step6 .....	30
Step7 .....	31
Step8 .....	31
4.3 Generation of version graph with node names .....	32
4.4 Isomorphism and Sample Input to the <i>gSpan</i> algorithm.....	32
Chapter 5 - Final analysis, Conclusion and Future Work.....	34
5.1 Experiment Analysis using <i>J48</i> .....	34
5.1.1 Baseline 0 analysis .....	34
5.1.2 Base line 1 analysis for Day6.....	35
5.1.3 Base line 2, Baseline 3, gSpan baseline analysis for Day6.....	36
5.2 Analysis using <i>LibSVM</i> .....	37
5.2.1 Baseline0 analysis .....	37
5.2.2 Baseline1, Baseline2, Baseline3 and gSpan base analysis after Day6 .....	37
5.3 Analysis using Multilayer Perceptron.....	38
5.3.1 Baseline0 analysis .....	38
5.3.2 Baseline1, Baseline2, Baseline3 and gSpan base analysis.....	39
5.3.3 Baseline3 with Augmented Data .....	39
5.4 Conclusion .....	40
5.5 Future Improvements .....	40

References.....	42
A.1 Arff Schema.....	43

## List of Figures

Figure 1.1 Sample Version Graph (edit graph).....	4
Figure 2.1 Example DoTTY graph .....	13
Figure 2.2 Weather data in a table format.....	15
Figure 2.3 Arff file format of weather data.....	16
Figure 3.1 Directory .....	20
Figure 3.2 Directory Structure .....	20
Figure 3.3 Version Graph .....	21
Figure 3.4 Sample version graph .....	22
Figure 4.1 Step1 of edit graph.....	27
Figure 4.2 Step2 of edit graph.....	27
Figure 4.3 Step3 of edit graph.....	28
Figure 4.4 Step4 of edit graph.....	29
Figure 4.5 Step5 of edit graph.....	30
Figure 4.6 Step6 of edit graph.....	30
Figure 4.7 Step7 of edit graph.....	31
Figure 4.8 Step8 of edit graph.....	31
Figure 4.9 Edit graph with node names .....	32
Figure 4.10 Isomorphism .....	33

## **Acknowledgements**

I thank my major professor Dr. William H. Hsu for guiding me during my Master's program. I am grateful to him for his suggestions and comments. I also appreciate his help in guiding my research. I also thank Dr. Gurdip Singh and Dr. Daniel Andresen for agreeing to serve on my Master's committee.

# Chapter 1 - Introduction

## 1.1 Problem Definition

*Wikipedia* is large scale collaborative editing project. This means that registered users all over the globe can edit the articles present in *Wikipedia*. Other than editing the articles, registered users can nominate articles for deletion. There are various factors that lead to this nomination like copyright violations, vandalism, advertising or other spam without relevant content, advertising or other spam without relevant content etc. When users edit an article in *Wikipedia*, the articles version by that user is stored in the history log of that article. We can construct an edit graph over a bunch of these edited versions. Edit graphs is discussed in Chapter 2.

In this thesis, I apply machine learning methods to determine how the final outcome of an AfD process is affected by factors such as the difference between versions of an article, number of edits, and number of disjoint edits (according to some contiguity constraints). My goal is to predict the outcome of an AfD which is nothing but the collaborative decision made by all the registered users for the AfD article log. This is done by analyzing the AfD page and editing history of the article. The technical objectives are to extract features from the AfD discussion and version history, as reflected in the edit history page, that reflect factors such as those discussed above, can be tested for relevance, and provide a basis for inductive generalization over past AfDs.

Here I first analyze the prediction in five different bases. The baseline0 includes analyzing the data which contains the number of votes (keep, merge and delete) for all the AfD articles. The baseline1 includes the data from baseline0 and also the difference between two edited versions of AfD articles. Baseline2 includes the data from baseline1 and also the number of edits since previous crawling. Baseline3 includes the data from baseline1 and also the number of disjoint edits since previous crawl. The number of disjoint edits is the number of nodes that appear in the edit graph. The final base, gSpan base includes finding the frequent sub graphs from group of edit graphs.

Frequent patterns are patterns (such as item sets, subsequences, or substructures) that appear in a dataset frequently. A substructure can refer to different structural forms, such as sub graphs, sub



trees, or sub lattices, which may be combined with item sets or subsequences. If a substructure occurs frequently, it is called a (frequent) structured pattern. Finding such frequent patterns plays an essential role in mining associations, correlations, and many other interesting relationships among data. Moreover, it helps in data classification, clustering, and other data mining tasks as well.

The first step of finding frequent sub graphs is constructing the edit graphs. Edit graph is a directed graph having nodes representing some textual data and edges represent the difference between the neighboring nodes. In this thesis, the textual data is one of the modified versions of an AfD article. The next step is then to find frequent sub graphs from a group of edit graphs. In my thesis “frequent patterns in *Wikipedia* pages”, I try using the features of edit graph to improve the overall prediction of outcome of the AfD articles.

For finding the frequent sub graphs I have used the *gSpan* algorithm. *gSpan* (graph-based Substructure pattern mining) discovers frequent substructures without candidate generation. *gSpan* builds a new lexicographic order among graphs, and maps each graph to a unique minimum DFS code as its canonical label. Based on this lexicographic order, *gSpan* adopts the depth-first search strategy to mine frequent connected sub graphs efficiently. Performance study shows that *gSpan* substantially outperforms previous algorithms, sometimes by an order of magnitude. Including the feature of frequent sub graphs to improve the overall prediction of outcome of AfD page can be considered for future improvements.

Machine learning tools are used to analyze the data from the AfD articles and check how accurate is the result (outcome of the AfD article) in accordance with the date related to the AfD article itself. One popular suite of machine learning tools, *Waikato Environment for Knowledge Analysis (Weka)* is used for this analysis. *Weka* provides implementations of learning algorithms that you can easily apply to your dataset. It also includes a variety of tools for transforming datasets, such as the algorithms for discretization. You can preprocess a dataset, feed it into a learning scheme, and analyze the resulting classifier and its performance all without writing any program code at all.

## **1.2 Goal and Technical Objectives**

The goal of this project is to analyze the data from the AfD articles and check how accurate is the result (outcome of the AfD article) in accordance with the date related to the AfD article

itself. The next step is to prove that including the features of edit graph and frequent sub graphs will improve the overall accuracy of the result with its respective data.

In order to achieve this goal, I have first extracted the AfD articles over a frequent period of time (fourteen versions over a period of seven days). Articles for deletion can be viewed through a few basic categories; it becomes much more accessible to the community at large. If a nominator is able to list an AfD correctly, they're probably able to tell whether it's a biography, media, science, and so on related topic. With a categorization option in place, editors are less likely to miss AfDs of interest, and save the time from browsing the entire list. The debates are sorted into the following subcategories using the indicated code in the `{{REMOVE THIS TEMPLATE WHEN CLOSING THIS AfD}}` template:

cat=M Media and music

cat=O Organization, corporation, or product

cat=B Biographical

cat=S Society topics

cat=W Web or internet

cat=G Games or sports

cat=T Science and technology

cat=F Fiction and the arts

cat=P Places and transportation

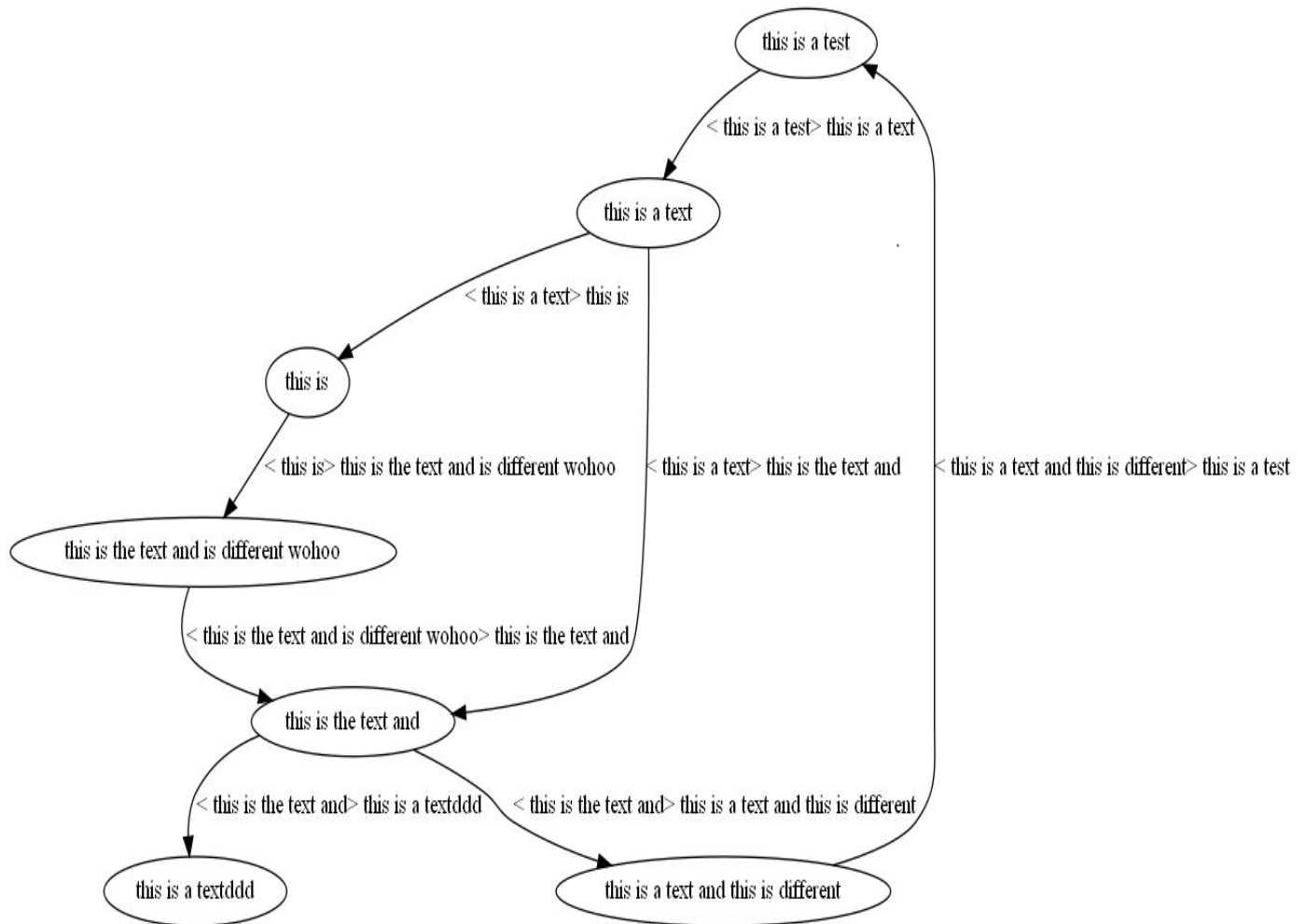
cat=I Indiscernible or unclassifiable topic

cat=? Nominator unsure of category (I have used 'Q' instead of '?' in this project)

cat=U Debate not yet sorted

The category name of a particular article is stored in the page titled "Editing *Wikipedia*:Articles for deletion/article\_name (section)". All the AfD articles are stored in the local system in a directory hierarchy. An example for one directory hierarchy is "M>List\_of\_Category\_III\_films\23-01-2011D". For each AfD article I store the History (directory), history, links, logs, talk, view AfD and the article itself. History contains the versions of the article since its first modification and first crawl. I have used Java programming language for this whole process.

After the AfD articles are stored in the local system, I construct a version graph for the articles stored in the History (directory). Nodes in the version graph represent the article content. The edge between two neighbors represents the difference (UNIX `diff`) between the two different versions of the articles connected by the edge. The UNIX `diff` is based on the LCS (Longest Common Subsequence) algorithm. The longest common subsequence (LCS) problem is to find the longest subsequence common to all sequences in a set of sequences (often just two). An example version graph looks like this:



**Figure 1.1 Sample Version Graph (edit graph)**

### ***1.2.1 UNIX `diff` command***

In *Unix* operating System, `diff` is a file comparison utility that outputs the differences between two files. It is typically used to show the changes between one version of a file and a former version of the same file. `diff` displays the changes made per line for text files.

```
> diff sample.doc vi.doc
```

If you only changed the name and date without moving any lines, your output should be similar to :-

```
18c18
```

```
< NAME DATE
```

```
---
```

```
> Melanie Johnson 10-12-96
```

For generating the graph I have used Java programming language and *Dotty*. *Dotty* is a graph editor for X Windows System. It may be run as a standalone editor, or as a front end for applications that use graphs. It can control multiple windows viewing different graphs.

Furthermore, each version graph is associated with a threshold. Here the threshold is the length of the difference. If the length of the difference between two versions is less than the threshold we merge the two versions or else they remain as two different versions of the graph.

In order to analyze the overall accuracy of the result of the AfD article with its corresponding date, we need to generate an *Arff* file which is given as input to the machine learning tool (*Weka*). The attribute-relation file format (*Arff*) is an input file format that is used in the *Weka* system.

For analyzing the overall result, the data is divided along 4 baselines. They are:-

### ***1.2.2 Baseline0***

Baseline0 includes the number of keep, delete and merge votes. It also has the end result (attribute class) of the AfD article. Here we analyze this data with various machine learning algorithms like *LibSVM*, *J48* and Multilayer Perceptron.

### ***1.2.3 Baseline1***

Baseline 1 includes six different data files. Each data file consists of data from Baseline0 and the difference between two versions of the AFD articles. The first data file consists of difference between Version 0 and Version1 of all the AFD articles plus the baseline0 data. The second data file consists of difference between Version 0 and Version3 of all the AFD articles

plus the baseline0 data. The third data file consists of difference between Version 0 and Version5 of all the AFD articles plus the baseline0 data. The same is the case for the remaining data files.

#### ***1.2.4 Baseline2***

Baseline 2 includes six different data files. Each data file consists of data from Baseline0 and the difference between two versions of the AFD articles and also the number of nodes difference between the nodes present in the History (directory) of each version. The first data file consists of difference between Version 0 and Version1 of all the AFD articles and the number of node difference (number of nodes in Version1 History *minus* the number of nodes in Version 0 History) and the baseline0 data. The second data file consists of difference between Version 0 and Version3 of all the AFD articles and the number of nodes difference (number of nodes in Version3 History *minus* the number of nodes in Version 0 History) and the baseline0 data. The same is the case for the remaining data files.

#### ***1.2.5 Baseline3***

Data in Baseline3 has the same format as in Baseline 2 but here we first draw a version graph on the nodes present in the History of each version of the AFD article. Since the edit graph has a cut off threshold the number of nodes present in the History directory decrease by the actual number because of merging.

#### ***1.2.6 gSpan Baseline***

Baseline 4 data includes the data from Baseline2. It also includes the node count which is derived from frequent sub graph discovery. This is done by first finding the frequent sub graphs of the edit graphs stored in the History (directory) of each article. Now we find how any nodes in each History (directory) are actually part of the frequent sub graphs. The sum of these nodes gives the node count. For example, the second data file includes the data present in the second data file from Baseline 2 and the count of nodes from History (directory) of version3 which appear in the frequent sub graph of that particular AFD article.

### 1.3 Overview

The following chapters will discuss about the background and related work, the algorithms used in order to implement this project, the end result and the future work for this project.

Chapter 2 discusses my literature review which includes a survey of the longest common subsequence (*LCS*) algorithm and the *gSpan* algorithm used for predicting the frequent patterns in the AfD articles. It also includes brief overview about graph plotting tool “dotty”, the machine learning tool *WEKA* used for data analysis and the input *Arff* file format which represents the data for Baseline0 through Baseline4.

Chapter 3 discusses my methodology, particularly how the algorithms discussed in Chapter 2 are used in this project. Here I also present the input data that is used to train a supervised inductive learning system to predict AfD outcomes. This training data is processed into a *WEKA*'s Attribute Relational File Format (*Arff*) for all feature sets used in my experiments. These include several experimental baselines and an augmented feature set based on the edit history, which I developed as part of the reported research.

Chapter 4 discusses about my experiment design. Here I present experimental features and the algorithms used to extract them from successive versions of a text file in a collaborative editing system. I use sample edit graphs graphs to illustrate these features and show how they are calculated from the original input data. This edit graph is the graph given as input to the *gSpan* algorithm. Also the alternative and independent measures survey to validate the original input data.

Chapter 5 discusses the results of my experiments using the experimental control, other baseline experiments, and the augmented feature set. I interpret the outcomes of these experiments and discuss how the empirical findings are in accordance with my hypothesis, and summarize how the implemented system relates to the overall problem definition. I conclude with a review of the original technical objectives and potential future work that can be done may yield more effective results and make the analytical process I have outlined in this thesis more efficient

## Chapter 2 - Background and Related work

### 2.1 Longest Common Subsequence Algorithm

String comparison is a central operation in various environments: a spelling error correction program tries to find the dictionary entry which resembles most a given word, in molecular biology, we want to compare two DNA or protein sequences to learn how homologous they are, in a file archive we want to store several versions of a source program compactly by storing only the original version as such; all other versions are constructed from the differences to the previous one, etc. An obvious measure for the closeness of two strings is to find the maximum number of identical symbols in them (preserving the symbol order).

This, by definition, is the longest common subsequence of the strings. Formally, we are comparing two input strings,  $X[1..m]$  and  $Y[1..n]$ , which are elements of the set  $C^*$ ; here  $C$  denotes the input alphabet containing  $T$  symbols. A subsequence

$S[1..s]$  of  $X[1..m]$  is obtained by deleting  $m - s$  from  $X$ . A common subsequence (cs) of  $X[1..m]$  and  $Y[1..n]$  - designated  $cs(X,Y)$  - is a subsequence which occurs in both strings ( we will assume that  $m \leq n$  ). The longest common subsequence (LCS) of strings  $X$  and  $Y$ ,  $lcs(X, Y)$ , is a common subsequence of maximal length. The length of  $lcs(X, Y)$  is denoted by  $r(X, Y)$ , or when the input strings are known, by  $r$ .

The lcs problem is a special case of the edit distance problem. The distance between  $X$  and  $Y$ ,  $Dist(X,Y)$ , is defined as the minimal number of elementary operations needed to transform the source string  $X$  to the target string  $Y$ . In practical applications, the operations are restricted to insertions, deletions and substitutions subjected to single input symbols. For each operation, an application dependent cost is assigned, whence the distance is defined as the minimum sum of transformation costs. Normalizing the costs of insertion and deletion to 1 and excluding the substitution operation (which can be accomplished by defining its cost to be  $\geq 2$ ), we have in fact the lcs problem: in order to transform  $X$  to  $Y$ , we first delete  $m - r$  symbols from the source to obtain  $lcs(X, Y)$ . To this string we insert  $n - r$  symbols to form the target.

Therefore,  $Dist(X,Y) = n+m-2.r(X,Y)$

Another closely related problem is the shortest common supersequence problem (scs), in which we want to find the shortest string containing  $X$  and  $Y$  as subsequences. In this case,

$r(X,Y) = m+n-r(\text{scs}(X,Y))$ , where  $r(\text{scs}(X, Y))$  denotes the length of the shortest common supersequence. After constructing the supersequence, we can extract the Ics in a linear scan through the scs. However, if the input consists of more than two strings ( $N$ -lcsmscs), there is no obvious symmetry between the problems anymore.

The Ics problem can be reduced to two other well-known problems also.  $\text{Lcs}(X,Y)$  is typically solved with the dynamic programming technique and filling an  $m \times n$  table. The table elements can be regarded as vertices in a graph and the simple dependencies between the table values define the edges. The task is to find the longest path between the vertices in the upper left and lower right corner of the table. Another reduction can be done to the longest increasing subsequence problem (lis). In the lis problem, we are given a sequence  $Z$  of  $z$  integers, and our task is to find longest subsequence of  $Z$  which is strictly increasing. The input to the lis problem is obtained from the Ics input as follows: take  $X[1]$  and find all positions  $j$  in  $Y$ , for which  $X[1] = Y[j]$ . Sort these indices into decreasing order. Build a sorted list for  $X[2]$  in a similar way and append it to the one we got for  $X[1]$ . Repeat this for all symbols in  $X$ . As a result, we have a sequence of integers having decreasing runs ('saw-tooth' form). Solving the lis of this sequence gives the  $Y$ -indices of those symbols which belong to the ICS.

The studies on theoretical complexity of the Ics problem give the lower bound  $\Omega(n^2)$ , if the elementary comparison operation is of type 'equal/unequal' and the alphabet size is unrestricted. However, if the input alphabet is fixed, we reach the lower bound  $\Omega(Tn)$ . In practice, the underlying encoding scheme for the symbols of the input alphabet implies a topological ordering between them and the  $<$ -comparison gives us more information reducing the lower bound to  $\Omega(n \log m)$ . Of the current methods, the  $O(n^2 / \log n)$  algorithm of Masek and Paterson is theoretically the fastest known.

The operation of diff is based on solving the longest common subsequence problem.

In this problem, you have two sequences of items:

a b c d f g h j q z

a b c d e f g i j k r x y z

and you want to find a longest sequence of items that is present in both original sequences in the same order. That is, you want to find a new sequence which can be obtained from the first sequence by deleting some items and from the second sequence by deleting other items. You also want this sequence to be as long as possible. In this case it is



a b c d f g j z

From a longest common subsequence it's only a small step to get `diff`-like output: if an item is absent in the subsequence but present in the original, it must have been deleted. (The '-' marks, below.) If it is absent in the subsequence but present in the second sequence, it must have been added in. (The '+' marks.)

e h i q k r x y  
+ - + - + + + +

## 2.2 Frequent pattern mining

Frequent patterns are patterns (such as itemsets, subsequences, or substructures) that appear in a data set frequently. For example, a set of items, such as milk and bread, which appear frequently together in a transaction data set is a frequent item set. A subsequence, such as buying first a PC, then a digital camera, and then a memory card, if it occurs frequently in a shopping history database, is a (frequent) sequential pattern. A substructure can refer to different structural forms, such as subgraphs, subtrees, or sublattices, which may be combined with itemsets or subsequences. If a substructure occurs frequently, it is called a (frequent) structured pattern.

### 2.2.1 *gSpan* (graph based substructure pattern mining)

*gSpan* (graph-based Substructure pattern mining) discovers frequent substructures without candidate generation. *gSpan* builds a new lexicographic order among graphs, and maps each graph to a unique minimum DFS code as its canonical label. Based on this lexicographic order, *gSpan* adopts the depth-rst search strategy to mine frequent connected subgraphs efficiently. Performance study shows that *gSpan* substantially outperforms previous algorithms, sometimes by an order of magnitude.

Frequent substructure pattern mining has been an emerging data mining problem with many scientific and commercial applications. As a general data structure, labeled graph can be used to model much complicated substructure patterns among data. Given a graph dataset,  $D = \{G_0, G_1, \dots, G_n\}$ ,  $\text{support}(g)$  denotes the number of graphs (in  $D$ ) in which  $g$  is a subgraph. The problem of frequent subgraph mining is to find any subgraph  $g$  s.t  $\text{support}(g) \geq \text{minSup}$  (a minimum support threshold). To reduce the complexity of the problem (meanwhile considering

the connectivity property of hidden structures in most situations), only frequent connected subgraphs are studied.

*gSpan* is the first algorithm that explores depth first search (DFS) in frequent subgraph mining. Two techniques, DFS lexicographic order and minimum DFS form a novel canonical labeling system to support DFS search. *gSpan* discovers all the frequent subgraphs without candidate generation and false positives pruning. It combines the growing and checking of frequent subgraphs into one procedure, thus accelerates the mining process.

*gSpan* uses a sparse adjacency list representation to store graphs. Algorithm 1 outlines the pseudo-code of the framework, which is self-explanatory (Note that  $\mathcal{G}$  represents the graph dataset,  $\mathcal{S}$  contains the mining result). Assume we have a label set  $\{ A, B, C, \dots \}$  for vertices, and  $\{ a, b, c, \dots \}$  for edges. In Algorithm 1 line 7-12, the first round will discover all the frequent subgraphs containing an edge  $A-A$  (with label alpha). The second round will discover all the frequent subgraphs containing  $A - B$  (with label alpha), but not any  $A-A$  (with label alpha) . This procedure repeats until all the frequent subgraphs are discovered. The database is shrunk when this procedure continues (Algorithm 1-line 10) and when the subgraph turns to be larger (Sub procedure 1-line 8, only graphs which contains this subgraph are considered. 'Ds' means the set of graphs in which 's' is a subgraph). Subgraph Mining is recursively called to grow the graphs and find all their frequent descendants. Subgraph Mining stops searching either when the support of a graph is less than  $\text{minSup}$ , or its code is not a minimum code, which means this graph and all its descendants have been generated and discovered before.

### ***Algorithm 1***

#### ***GraphSet Projection(D,S).***

- 1: sort the labels in  $\mathcal{L}$  by their frequency;
- 2: remove infrequent vertices and edges;
- 3: relabel the remaining vertices and edges;
- 4:  $S_1 \leftarrow$  all frequent 1-edge graphs in  $D$ ;
- 5: sort  $S_1$  in DFS lexicographic order;
- 6:  $S \leftarrow S_1$ ;
- 7: for each edge  $e$  belongs to  $S_1$  do

```

8: initialize s with e, set s.D by graphs which contains e;
9: Subgraph Mining( $\check{D}, S, s$ );
10:  $\check{D} \leftarrow D - e$ ;
11: if  $|D| < \text{minSup}$  ;
12: break;

```

### ***Subprocedure 1***

#### ***Subgraph Mining( $\check{D}, S, s$ ).***

```

1: if  $s \neq \text{min}(s)$ 
2:  $\triangleright$  return;
3:  $S \leftarrow S \cup \{s\}$ 
4: enumerate s in each graph in  $\check{D}$  and count its children;
5: for each c, c is s' child do
6: if  $\text{support}(c) \geq \text{minSup}$ 
7:    $s \leftarrow c$ ;
8: Subgraph Mining( $\check{D}_s, S, s$ );

```

## **2.3 Dotty (directed graph editor)**

DOT is a Unix filter that takes a DOT language file specifying the objects in a directed graph and output a optimal (in some sense) layout of the objects in one of a number of formats including Postscript and Maker Interchange Format (MIF). DOTTY is a customizable directed graph editor. It is a X Windows based GUI that is very easy to use. With just a few clicks of the mouse, you can build a graph. DOTTY is built upon DOT.

These lines in a file:

```

digraph G {
  a -> b [label="hello", style=dashed];
  a -> c [label="world"];
  c -> d; b -> c; d -> a;
  b [shape=Mdiamond, label="this is b"];

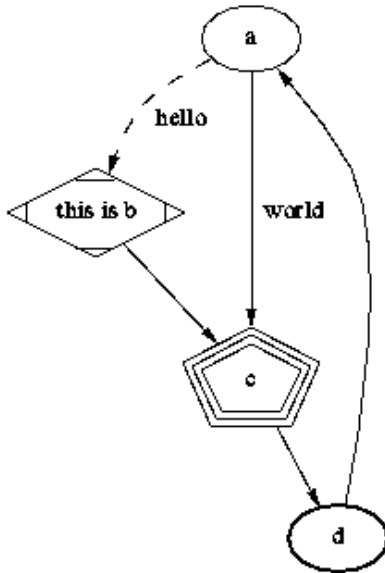
```

```

c [shape=polygon, sides=5, peripheries=3];
d [style=bold];
}

```

Produce this:-



**Figure 2.1 Example DoTTy graph**

### 2.3 Counting the number of merge, keep and delete votes

Users of *Wikipedia* pages can nominate an article for deletion by placing `{{subst:proposed deletion}}` on the page. If any person (even the author him/herself) objects to the deletion (usually by removing the `{{proposed deletion}}` tag), the proposal is aborted and may not be re-proposed. The article is first checked and then deleted by an administrator 7 days after nomination. It may be undeleted upon request.

If the article is undeleted it is either kept as a separate page or merged into some other page. If the article is merged, the administrator places `{{afd-mergeto}}` and `{{afd-mergefrom}}` on the appropriate pages, which tags the articles for merging via ordinary editing.

If the article is kept, the administrator includes "Keep" in the edit section of the page.

By counting the number of keep, delete and merge votes of the AfD articles during frequent time intervals over a period of time, we can analyze the frequent patterns in *Wikipedia* pages.

## 2.4 Preparing the input

Preparing input for a data mining investigation usually consumes the bulk of the effort invested in the entire data mining process. Here we look at a particular input file format, the attribute-relation file format (*Arff*), that is used in the *WEKA* system.

### 2.4.1 ARFF Format

We look at a standard way of representing datasets, called an ARFF file. We describe the regular version, but there is also a version called XRFF, which, as the name suggests, gives ARFF header and instance information in the extensible Markup Language (XML). Figure 2.3 shows an ARFF file for the weather data in Table 2.2, the version with some numeric features. Lines beginning with a % sign are comments. Following the comments at the beginning of the file are the name of the relation (*weather*) and a block defining the attributes (*outlook*, *temperature*, *humidity*, *windy*, *play?*). Nominal attributes are followed by the set of values they can take on, enclosed in curly braces. Values can include spaces; if so, they must be placed within quotation marks. Numeric values are followed by the keyword *numeric*. Although the weather problem is to predict the class value '*play?*' from the values of the other attributes, the class attribute is not distinguished in any way in the data file. The ARFF format merely gives a dataset; it does not specify which of the attributes the one that is supposed to be predicted is. This means that the same file can be used for investigating how well each attribute can be predicted from the others, or it can be used to find association rules or for clustering.

Outlook	Temperature	Humidity	Windy	Play Time
Sunny	85	85	false	5
Sunny	80	90	true	0
Overcast	83	86	false	55
Rainy	70	96	false	40
Rainy	68	80	false	65
Rainy	65	70	true	45
Overcast	64	65	true	60
Sunny	72	95	false	0
Sunny	69	70	false	70
Rainy	75	80	false	45
Sunny	75	70	true	50
Overcast	72	90	true	55
Overcast	81	75	false	75
Rainy	71	91	true	10

**Figure 2.2 Weather data in a table format**

```
weather.arff
1 @relation weather
2
3 @attribute outlook {sunny, overcast, rainy}
4 @attribute temperature real
5 @attribute humidity real
6 @attribute windy {TRUE, FALSE}
7 @attribute play {yes, no}
8
9 @data
10 sunny, 85, 85, FALSE, no
11 sunny, 80, 90, TRUE, no
12 overcast, 83, 86, FALSE, yes
13 rainy, 70, 96, FALSE, yes
14 rainy, 68, 80, FALSE, yes
15 rainy, 65, 70, TRUE, no
16 overcast, 64, 65, TRUE, yes
17 sunny, 72, 95, FALSE, no
18 sunny, 69, 70, FALSE, yes
19 rainy, 75, 80, FALSE, yes
20 sunny, 75, 70, TRUE, yes
21 overcast, 72, 90, TRUE, yes
22 overcast, 81, 75, FALSE, yes
23 rainy, 71, 91, TRUE, no
24
```

**Figure 2.3** Arff file format of weather data

Following the attribute definitions is an @data line that signals the start of the instances in the dataset. Instances are written one per line, with values for each attribute in turn, separated by commas. If a value is missing, it is represented by a single question mark (there are no missing values in this dataset). The attribute specifications in ARFF files allow the dataset to be checked to ensure that it contains legal values for all attributes, and programs that read ARFF files do this checking automatically. As well as nominal and numeric attributes, exemplified by the weather data, the ARFF format has three further attribute types: string attributes, date attributes, and relation-valued attributes. String attributes have values that are textual. Suppose you have a string attribute that you want to call description. In the block defining the attributes it is specified like this:

*@attribute description string*

Then, in the instance data, include any character string in quotation marks (to include quotation marks in your string, use the standard convention of preceding each one by a backslash, \).

Strings are stored internally in a string table and represented by their address in that table. Thus, two strings that contain the same characters will have the same value. String attributes can have values that are very long—even a whole document. To be able to use string attributes for text mining, it is necessary to be able to manipulate them. For example, a string attribute might be converted into many numeric attributes, one for each word in the string, whose value is the number of times that word appears.

## **2.5 Brief overview of WEKA**

Experience shows that no single machine learning scheme is appropriate to all data mining problems. The universal learner is an idealistic fantasy. As we have emphasized throughout this book, real datasets vary, and to obtain accurate models the bias of the learning algorithm must match the structure of the domain. Data mining is an experimental science.

The *WEKA* workbench is a collection of state-of-the-art machine learning algorithms and data preprocessing tools. It includes virtually all the algorithms described in this book. It is designed so that you can quickly try out existing methods on new datasets in flexible ways. It provides extensive support for the whole process of experimental data mining, including preparing the input data, evaluating learning schemes statistically, and visualizing the input data and the result of learning. As well as a variety of learning algorithms, it includes a wide range of preprocessing tools. This diverse and comprehensive toolkit is accessed through a common interface so that its users can compare different methods and identify those that are most appropriate for the problem at hand.

*WEKA* provides implementations of learning algorithms that you can easily apply to your dataset. It also includes a variety of tools for transforming datasets, such as the algorithms for discretization. You can preprocess a dataset, feed it into a learning scheme, and analyze the resulting classifier and its performance all without writing any program code at all. I have used *J48*, *LibSVM* and *Multilayer perceptron* for the analysis of my data. In my project I have used the Explorer to analyze the data. More information about using *WEKA* and different classifiers can be found in *Data Mining - Practical Machine Learning Tools and Techniques*, Third Edition by Ian H. Witten, Eibe Frank and Mark A. Hall.



## **2.6 Machine Learning Algorithms Used For Classification**

In machine learning and pattern recognition, classification refers to an algorithmic procedure for assigning a given piece of input data into one of a given number of categories. An example would be assigning a given email into "spam" or "non-spam" classes or assigning a diagnosis to a given patient as described by observed characteristics of the patient (gender, blood pressure, presence or absence of certain symptoms, etc.). An algorithm that implements classification, especially in a concrete implementation, is known as a classifier. The term "classifier" sometimes also refers to the mathematical function, implemented by a classification algorithm that maps input data to a category.

### ***2.6.1 Classifier used in this project***

#### ***2.6.1.1 C4.5 Algorithm***

C4.5 is an algorithm used to generate a decision tree developed by Ross Quinlan. C4.5 is an extension of Quinlan's earlier ID3 algorithm. The decision trees generated by C4.5 can be used for classification, and for this reason, C4.5 is often referred to as a statistical classifier.

C4.5 builds decision trees from a set of training data in the same way as ID3, using the concept of information entropy. The training data is a set  $S = s_1, s_2, \dots$  of already classified samples. Each sample  $s_i = x_1, x_2, \dots$  is a vector where  $x_1, x_2, \dots$  represent attributes or features of the sample. The training data is augmented with a vector  $C = c_1, c_2, \dots$  where  $c_1, c_2, \dots$  represent the class to which each sample belongs.

At each node of the tree, C4.5 chooses one attribute of the data that most effectively splits its set of samples into subsets enriched in one class or the other. Its criterion is the normalized information gain (difference in entropy) that results from choosing an attribute for splitting the data. The attribute with the highest normalized information gain is chosen to make the decision. The C4.5 algorithm then recurs on the smaller sub lists.

*J48* is an open source Java implementation of the C4.5 algorithm in the *WEKA* data mining tool.

#### ***2.6.1.2 Support vector machine***

A support vector machine (SVM) is a concept in computer science for a set of related supervised learning methods that analyze data and recognize patterns, used for classification and regression analysis. The standard SVM takes a set of input data and predicts, for each given input, which of two possible classes the input is a member of, which makes the SVM a non-probabilistic binary linear classifier. Given a set of training examples, each marked as belonging to one of two categories, an SVM training algorithm builds a model that assigns new examples into one category or the other. An SVM model is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible. New examples are then mapped into that same space and predicted to belong to a category based on which side of the gap they fall on.

More formally, a support vector machine constructs a hyperplane or set of hyperplanes in a high- or infinite- dimensional space, which can be used for classification, regression, or other tasks. Intuitively, a good separation is achieved by the hyperplane that has the largest distance to the nearest training data points of any class (so-called functional margin), since in general the larger the margin the lower the generalization error of the classifier. *LibSVM* is a library of SVMs which is actively patched.

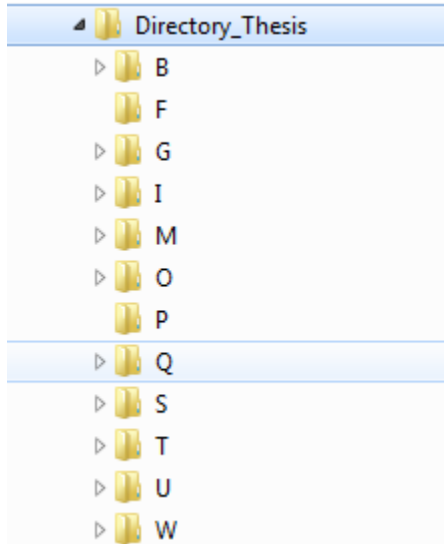
### ***2.6.1.3 Multilayer Perceptron***

A multilayer perceptron (MLP) is a feed forward artificial neural network model that maps sets of input data onto a set of appropriate output. An MLP consists of multiple layers of nodes in a directed graph, with each layer fully connected to the next one. Except for the input nodes, each node is a neuron (or processing element) with a nonlinear activation function. MLP utilizes a supervised learning technique called back propagation for training the network. Back Propagation is a machine learning algorithm to train a neural network. MLP is a modification of the standard linear perceptron, which can distinguish data that is not linearly separable.

## Chapter 3 - Methodology

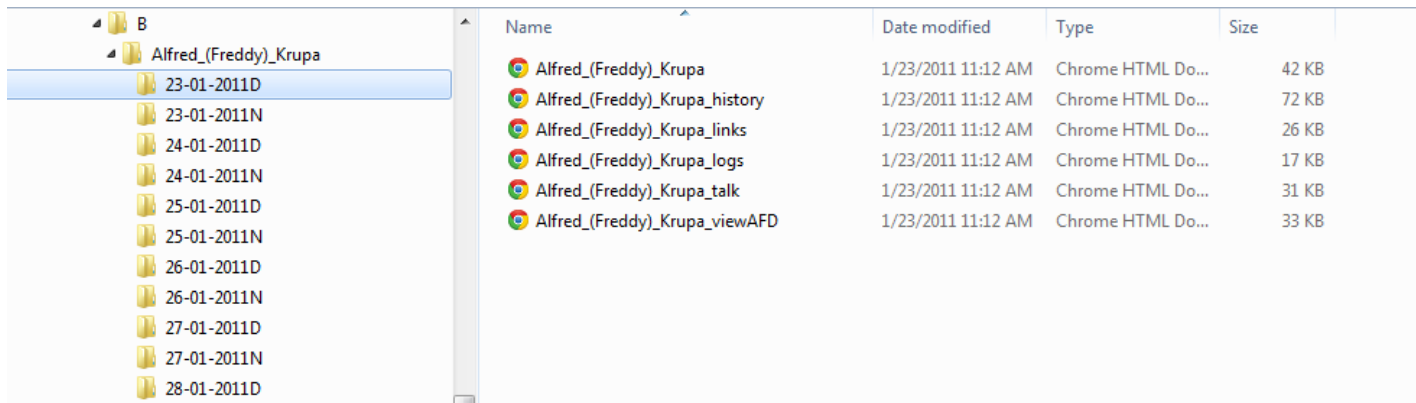
### 3.1 Extracting the AfD pages into the local system

For extracting the AfD pages into the local system I have used Java programming. These AfD pages are stored in directory hierarchy.



**Figure 3.1 Directory**

The directory hierarchy includes Directory as the root node. On the first level we have the category nodes 'B', 'F', 'G', 'I', 'M', 'O', 'P', 'Q', 'S', 'T', 'U', 'W'. Each of these categories includes the wiki AfD articles directory. Furthermore, each of these AfD article directories includes the different version for that article, where each version includes the AfD article, its history, its links, logs, talk and view\_AfD page of that article.

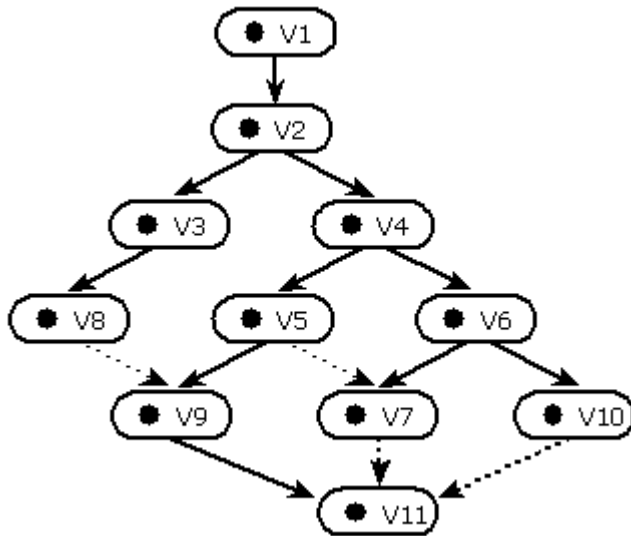


**Figure 3.2 Directory Structure**

For storing all the above information into the local system, I have used JAVA StringTokenizer class and its operations.

### 3.2 Generating version graph from AfD articles

AfD article's nodes stored in each versions History directory are given as input to the program which generates the version graph. A version graph indicates how the various versions relate to each other. An object's version graph consists of nodes and arrows. Each node represents a version of the object and each arrow points from one object version to a successor of that object version. The following figure shows a typical version graph for an object with 11 versions.

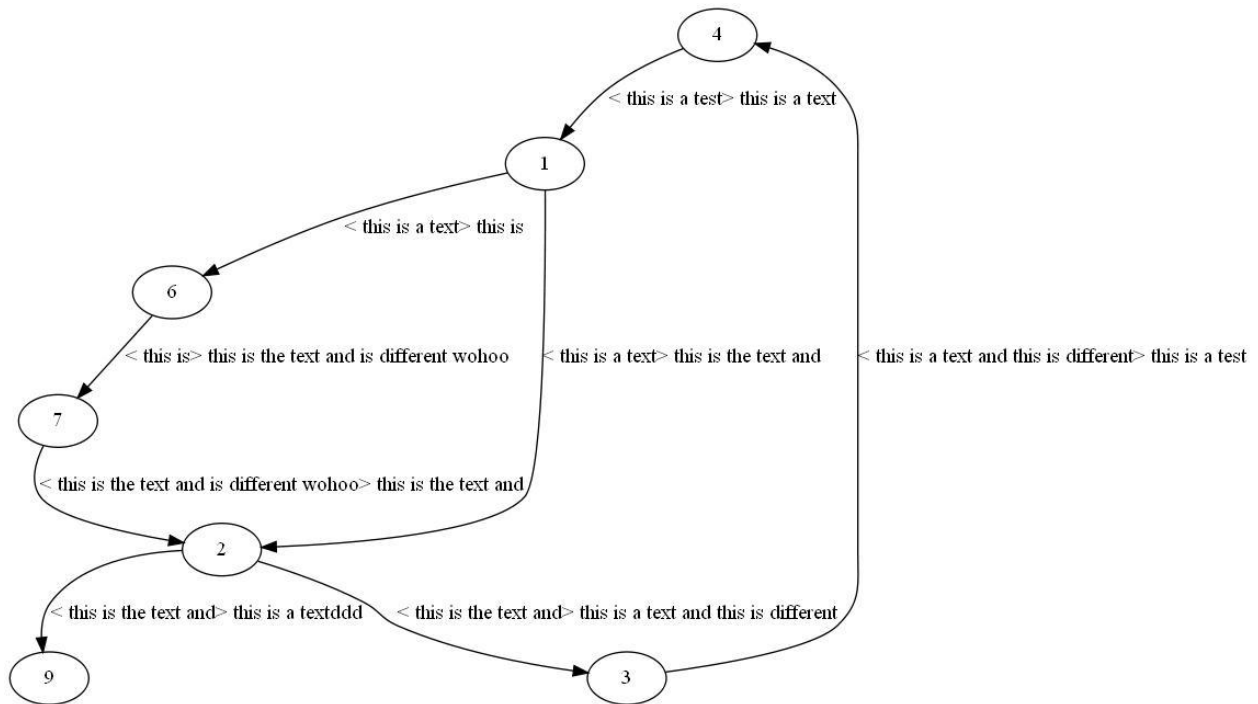


**Figure 3.3 Version Graph**

There are two kinds of arrows. A solid arrow indicates the creation of one object version based on another. For example, the solid arrow from Version 6 to Version 7 indicates that Version 7 was created based on Version 6. A dashed arrow indicates the merging of property values and collections from one object into another. For example, the dashed arrow from Version 10 to Version 11 indicates that property values and collections from Version 10 were merged into Version 11.

For simplicity, the version graph that the program generates only includes the solid edges and we merge the versions with edge value less than the threshold into a single version.

The edges in the version graph represent the UNIX difference between two versions as described in chapter 1. A sample version graph looks like this:



**Figure 3.4 Sample version graph**

These set of version graphs are given as input to the *gSpan* algorithm which is used for frequent pattern recognition.

### 3.3 Input format to *gSpan* Algorithm

The *gSpan* program is written in C++ and takes input format as follows:

"t # N" means the Nth graph,

"v M L" means that the Mth vertex in this graph has label L,

"e P Q L" means that there is an edge connecting the Pth vertex with the Qth vertex. The edge has label L.

M, N, P, Q, and L are integers.

The output of the *gSpan* algorithm has the format as follows:

t # id \* support

vertex-edge list, same as the input format

x graph\_id list

"id" is an integer, the serial number of the pattern.

"support" is the absolute frequency of the graph pattern.

"graph\_id list" is a list of graphs that contain the pattern.

Example:

```
gSpan -f Compound_422 -s 0.1 -o -i
```

Which asks *gSpan* to discover all frequent sub graphs whose frequency is 10%. Here, the input file is "Chemical\_340". It outputs the patterns in "Chemical\_340.fp", with graph id.

## 3.4 Input format for Arff data files

### 3.4.1 *Baseline0*

The input format for *baseline1* includes the attributes

@relation Base0

@attribute Keep numeric

@attribute Delete numeric

@attribute Merge numeric

@attribute Class { keep, merge, delete }

@data

DATA HERE

### 3.4.2 *Baseline1*

The input format for *baseline1* includes the attributes

@relation DayX

@attribute Difference numeric

@attribute Keep numeric

@attribute Delete numeric

@attribute Merge numeric

@attribute Class { keep, merge, delete }

@data

Data Here

### ***3.4.3 Baseline2***

The input format for baseline2 includes the attributes

@relation DayX

@attribute Difference numeric

@attribute Number\_of\_nodes\_Difference numeric

@attribute Keep numeric

@attribute Delete numeric

@attribute Merge numeric

@attribute Class { keep, merge, delete }

@data

Data Here

### ***3.4.4 Baseline3***

The input format for baseline3 includes the attributes

@relation DayX

@attribute Difference numeric

@attribute Number\_of\_nodes\_Merged numeric

@attribute Keep numeric

@attribute Delete numeric

@attribute Merge numeric

@attribute Class { keep, merge, delete }

@data

Data Here

### ***3.4.5 gSpan Baseline***

@relation DayX

@attribute Difference numeric

@attribute Frequent\_subgraphOccurence numeric

@attribute Keep numeric

@attribute Delete numeric

@attribute Merge numeric

@attribute Class { keep, merge, delete }

@data

Data Here



## Chapter 4 - Experimental Design

### 4.1 Sample text files given as initial input

Since it is harder to analyze the vertex graph for the AfD articles, I have used sample text files. Each of these sample text file is considered as a different version of its present version. These files are number 1 through 9. Where 1 denotes the first version and 9 is the last version. These set of files is give as the input to the methodology discussed in section 3.2 (generating version graph).

### 4.2 Step by Step Generation of Version graph for sample text files:

The sample text files have the following information

File 1:

this is a text

File 2:

this is the text and

File 3:

this is a text and this is different

File 4:

this is a test

File 5:

this is a text

File 6:

this is

File 7:

this is the text and is different wohoo

File 8:

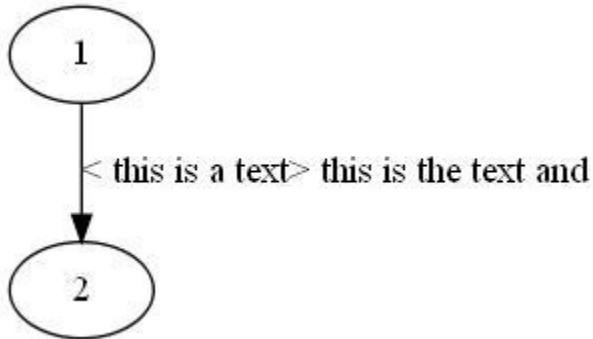
this is the text and

File 9:

this is a textddd

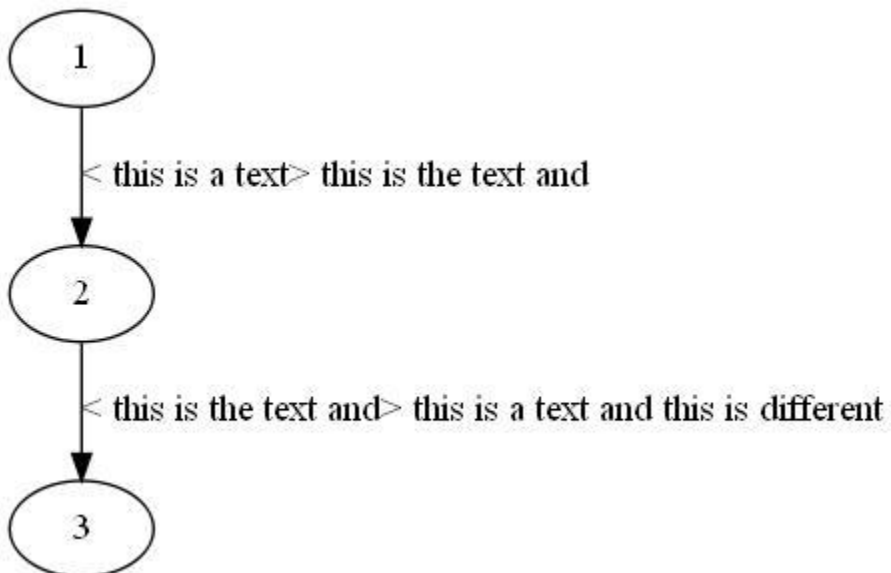
During each step if a version is same as its previous version we merge both the versions into a single version which is numbered after the minimum of the two versions merged.

*Step1*



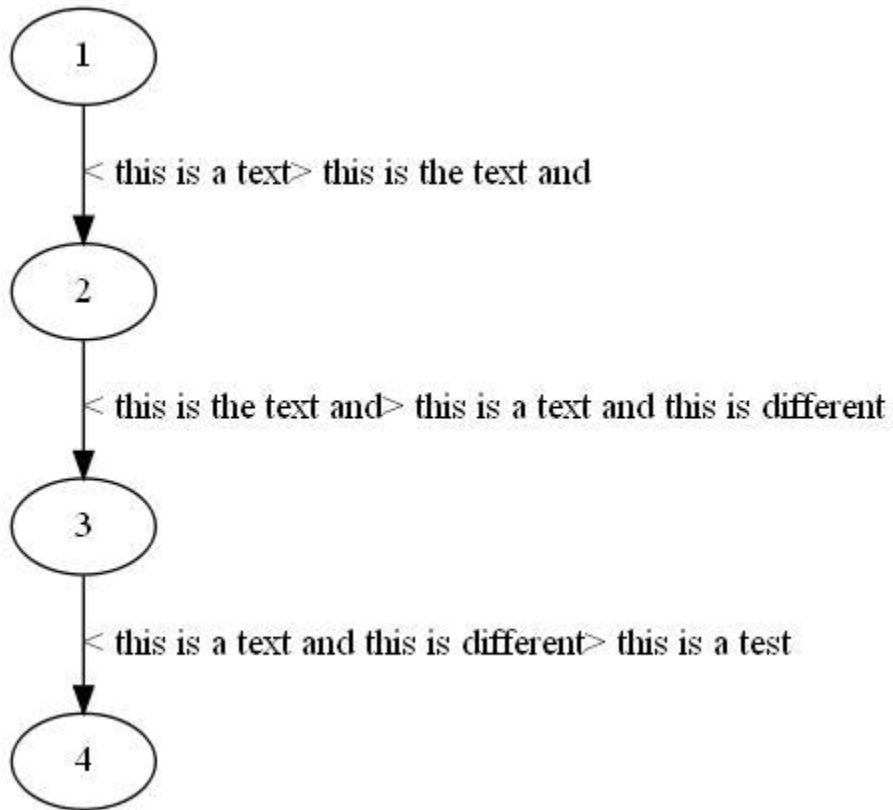
**Figure 4.1 Step1 of edit graph**

*Step2*



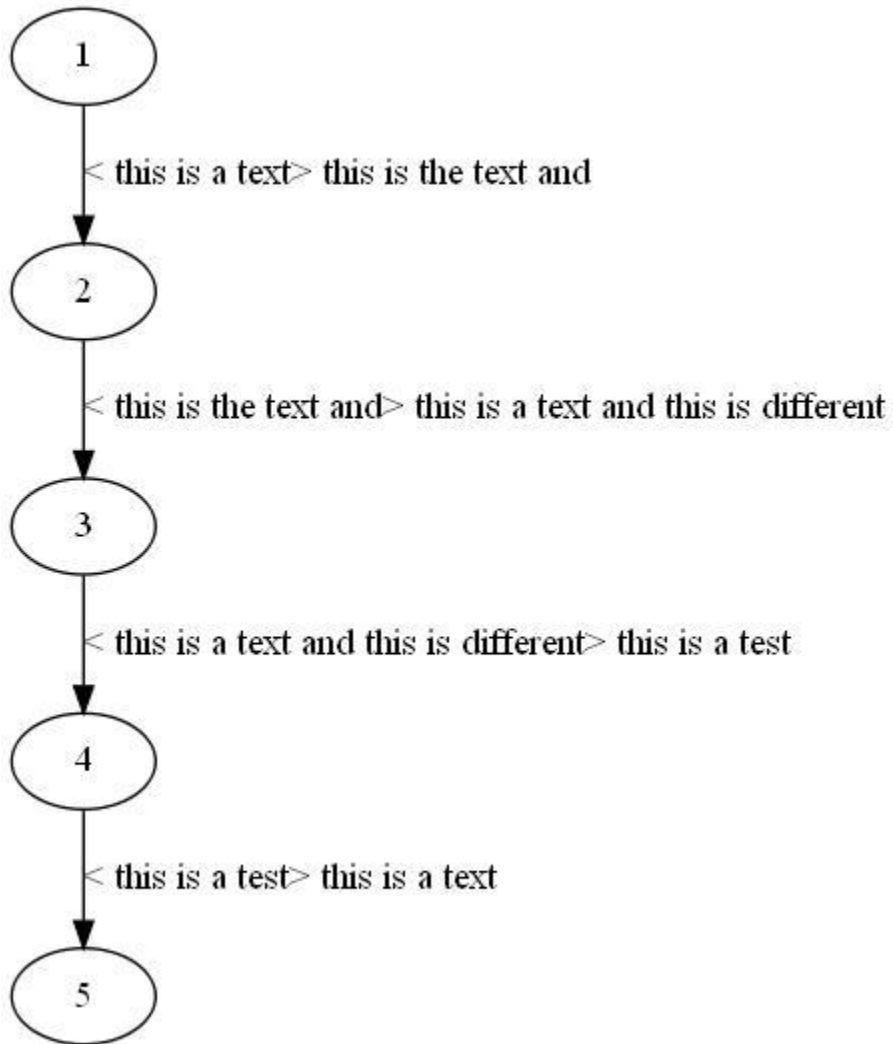
**Figure 4.2 Step2 of edit graph**

*Step3*



**Figure 4.3 Step3 of edit graph**

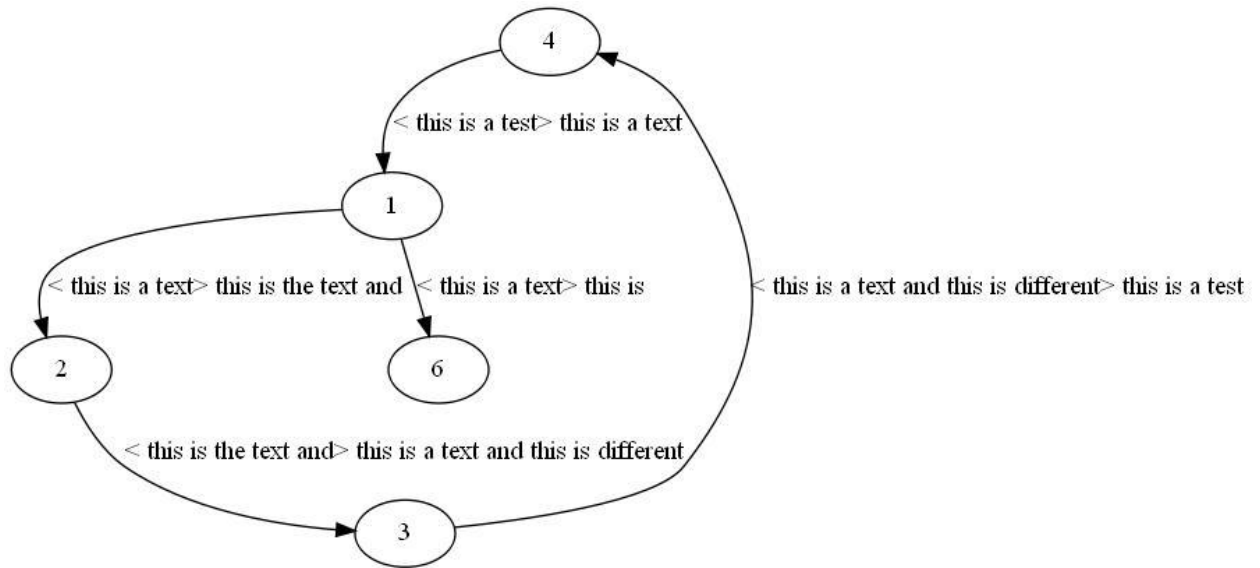
### Step4



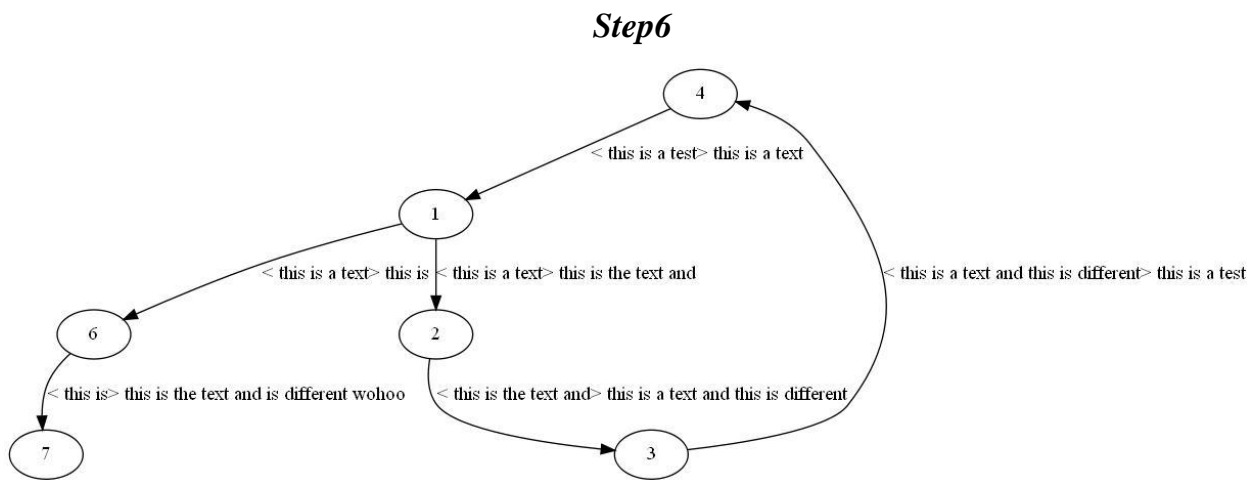
**Figure 4.4 Step4 of edit graph**

### Step5

Since, version 5 (file 5) is same as node 1 we merge it with version1. By merging these two nodes, node 4 becomes the parent of node 1.



**Figure 4.5 Step5 of edit graph**



**Figure 4.6 Step6 of edit graph**

### Step7

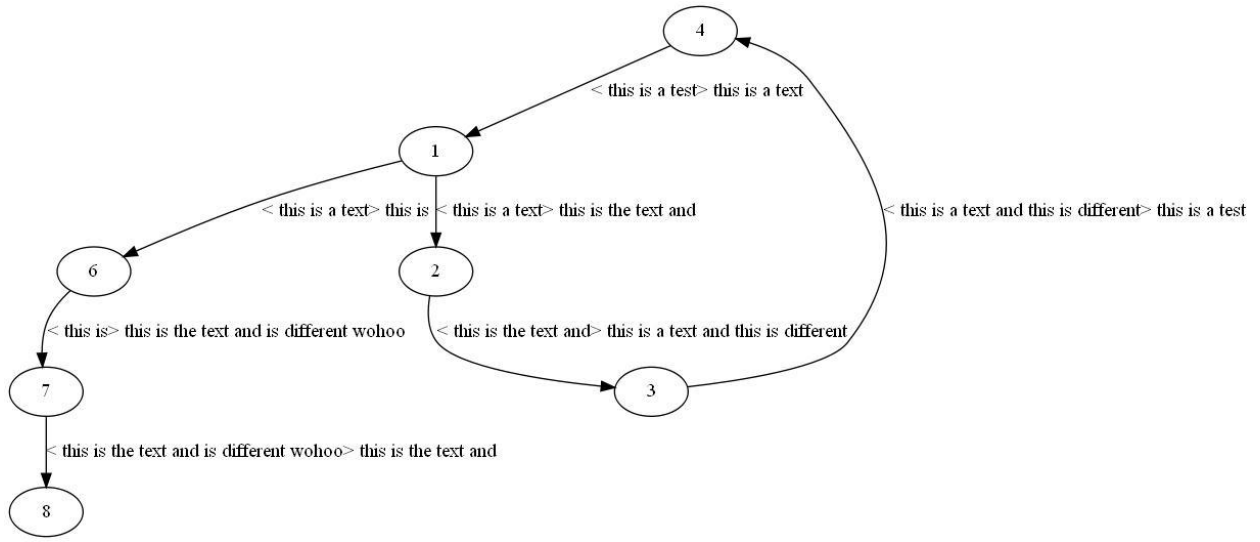


Figure 4.7 Step7 of edit graph

### Step8

Since, version 8 (file 8) is same as node 2 we merge it with version2. By merging these two nodes, node 7 becomes the parent of node 2.

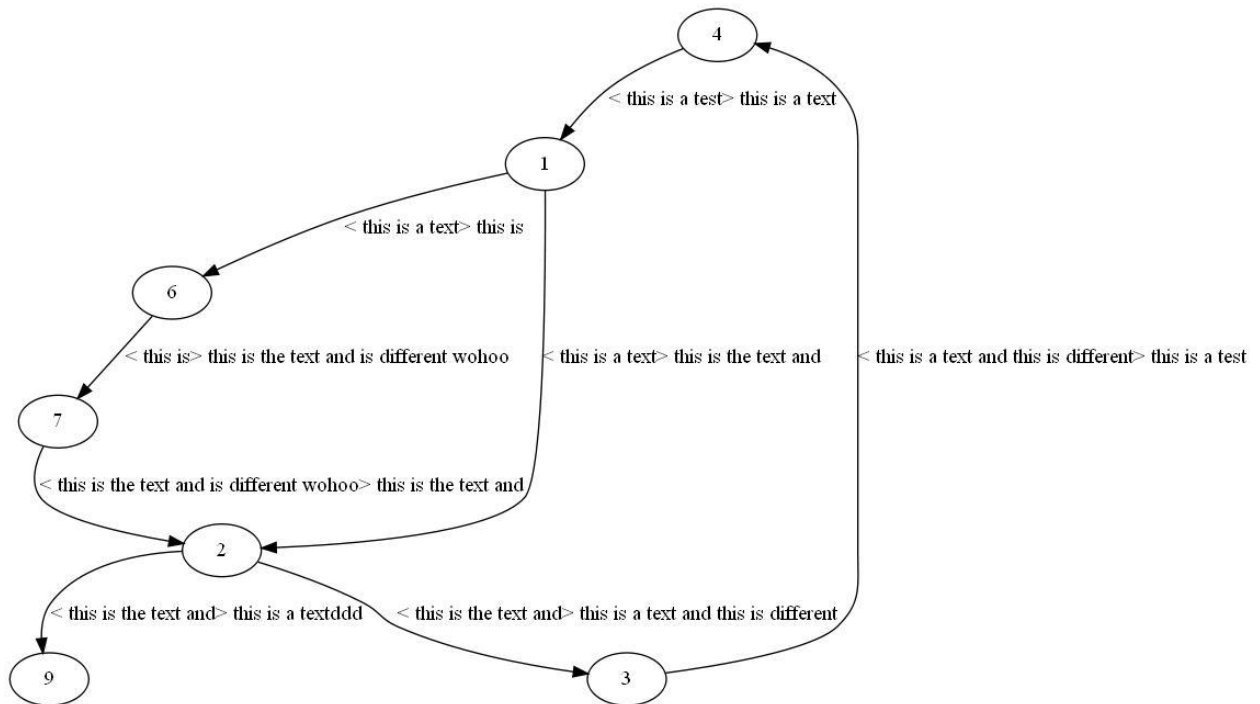


Figure 4.8 Step8 of edit graph

### 4.3 Generation of version graph with node names

The below version graph is same as the version graph in step 8 of section 4.2 but with node names.

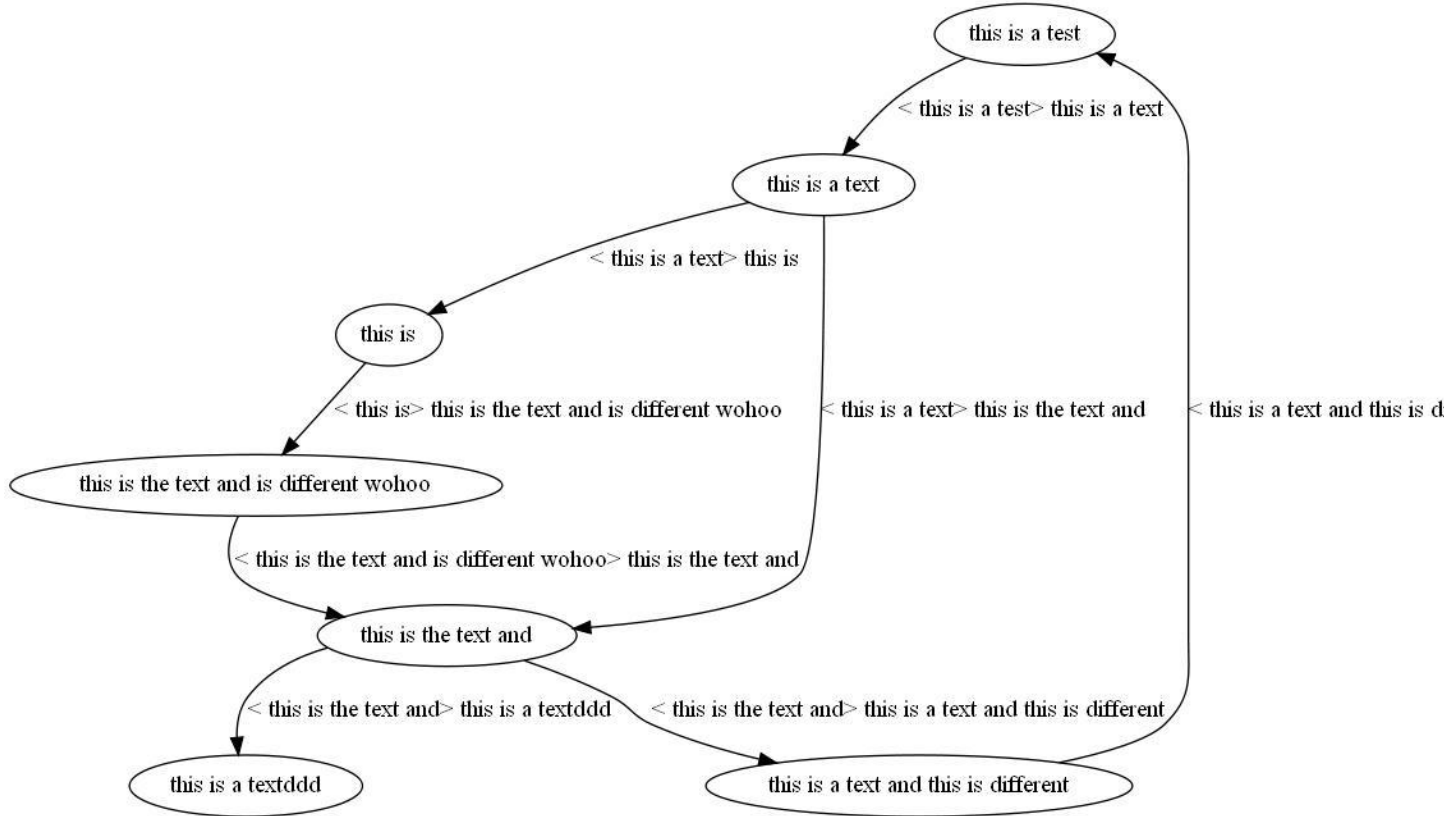


Figure 4.9 Edit graph with node names

### 4.4 Isomorphism and Sample Input to the *gSpan* algorithm

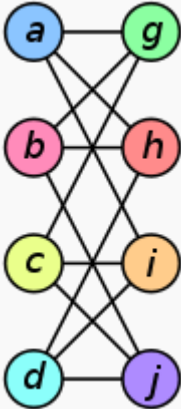
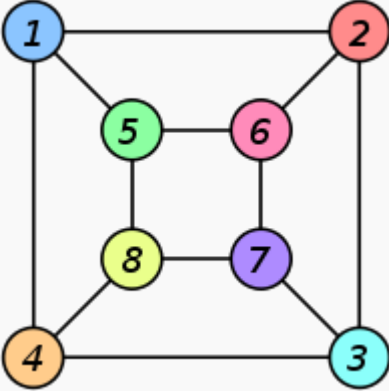
In graph theory, an isomorphism of graphs  $G$  and  $H$  is a bijection between the vertex sets of  $G$  and  $H$  such that any two vertices  $u$  and  $v$  of  $G$  are adjacent in  $G$  if and only if  $f(u)$  and  $f(v)$  are adjacent in  $H$ . This kind of bijection is commonly called "edge-preserving bijection", in accordance with the general notion of isomorphism being a structure-preserving bijection.

In the above definition, graphs are understood to be undirected non-labeled non-weighted graphs. However, the notion of isomorphism may be applied to all other variants of the notion of graph, by adding the requirements to preserve the corresponding additional elements of structure: arc directions, edge weights, etc., with the following exception. When spoken about graph labeling

with unique labels, commonly taken from the integer range  $1, \dots, n$ , where  $n$  is the number of the vertices of the graph, two labeled graphs are said to be isomorphic if the corresponding underlying unlabeled graphs are isomorphic.

If an isomorphism exists between two graphs, then the graphs are called isomorphic and we write. In the case when the bijection is a mapping of a graph onto itself, i.e., when  $G$  and  $H$  are one and the same graph, the bijection is called an automorphism of  $G$ .

The graph isomorphism is an equivalence relation on graphs and as such it partitions the class of all graphs into equivalence classes. A set of graphs isomorphic to each other is called an isomorphism class of graphs.

Graph G	Graph H	An isomorphism between G and H
		$f(a) = 1$ $f(b) = 6$ $f(c) = 8$ $f(d) = 3$ $f(g) = 5$ $f(h) = 2$ $f(i) = 4$ $f(j) = 7$

**Figure 4.10 Isomorphism**



## Chapter 5 - Final analysis, Conclusion and Future Work

In this chapter I analyze the *WEKA* classifier output for the input data presented in Chapter 3. The results show that adding the factors other than the number of votes like difference between versions of an article, number of edits, and number of disjoint edits (according to some contiguity constraints) does not help much in the prediction process. This is mainly because of the insufficient data I have crawled from the AfD page. The results may vary significantly for large amounts of data.

The data crawled in this project is the AfD log as on August 1, 2011 ([http://en.wikipedia.org/wiki/Wikipedia:Articles\\_for\\_deletion/Log/2011\\_August\\_1#Suicide\\_by\\_hanging](http://en.wikipedia.org/wiki/Wikipedia:Articles_for_deletion/Log/2011_August_1#Suicide_by_hanging)). There are 64 instances of AfD articles from this log. I have done the experimental analysis over these 64 instances. In the following chapter I discuss the various outputs for the four baselines with *J48*, *LibSVM* and *multilayer perceptron* classifiers.

### 5.1 Experiment Analysis using *J48*

#### 5.1.1 Baseline 0 analysis

*J48* pruned tree for:

Keep  $\leq 1$

| Merge  $\leq 0$

| | Delete  $\leq 0$ : keep (2.0)

| | Delete  $> 0$ : delete (36.0)

| Merge  $> 0$ : merge (3.0)

Keep  $> 1$

| Delete  $\leq 4$ : keep (20.0/2.0)

| Delete  $> 4$ : delete (3.0)

Number of Leaves : 5

Size of the tree : 9

I have analyzed the output by looking at the confusion matrix and ROC weighted area below. It clearly shows that *J48* does not give an accurate prediction. It has 4 false negatives and 5 false positives.

	TP Rate	FP Rate	Precision	Recall	F-Measure	ROC Area	Class
	0.8	0.114	0.762	0.8	0.78	0.88	keep
	1	0.016	0.75	1	0.857	0.992	merge
	0.878	0.13	0.923	0.878	0.9	0.917	delete
Weighted Avg.	0.859	0.12	0.865	0.859	0.861	0.909	

==== Confusion Matrix ====

a b c <-- classified as

16 1 3 | a = keep

0 3 0 | b = merge

5 0 36 | c = delete

### ***5.1.2 Base line 1 analysis for Day6***

*J48* pruned tree

-----

Keep <= 1

| Merge <= 0

| | Delete <= 0: keep (2.0)

| | Delete > 0: delete (36.0)

| Merge > 0: merge (3.0)

Keep > 1

| Delete <= 4: keep (20.0/2.0)

| Delete > 4: delete (3.0)

Number of Leaves : 5

Size of the tree : 9

The output confusion matrix and ROC weighted area is shown below. It clearly shows that *J48* still does not give an accurate prediction. It has 4 false negatives and 5 false positives. There is no improvement over Baseline0

=== Detailed Accuracy By Class ===

TP Rate	FP Rate	Precision	Recall	F-Measure	ROC Area	Class
0.8	0.114	0.762	0.8	0.78	0.88	keep
1	0.016	0.75	1	0.857	0.992	merge
0.878	0.13	0.923	0.878	0.9	0.917	delete
Weighted Avg.	0.859	0.12	0.865	0.859	0.861	0.909

=== Confusion Matrix ===

a b c <-- classified as

16 1 3 | a = keep

0 3 0 | b = merge

5 0 36 | c = delete

### ***5.1.3 Base line 2, Baseline 3, gSpan baseline analysis for Day6***

Baseline 2, 3 and 4 don't have any improvement over the Baseline0. This clearly means that *J48* algorithm is not accurate for data I have for analysis.

## 5.2 Analysis using *LibSVM*

### 5.2.1 *Baseline0 analysis*

Analysis using *LibSVM* shows that there is improvement over the process of prediction of the AFD page. The results in the confusion matrix clearly shows that there are only 2 false positives and 1 false negative which is much accurate than analysis using *J48*.

TP Rate	FP Rate	Precision	Recall	F-Measure	ROC Area	Class
0.95	0.045	0.905	0.95	0.927	0.952	keep
1	0	1	1	1	1	merge
0.951	0.043	0.975	0.951	0.963	0.954	delete
Weighted Avg.	0.953	0.042	0.954	0.953	0.953	0.956

==== Confusion Matrix ====

a b c <-- classified as

19 0 1 | a = keep

0 3 0 | b = merge

2 0 39 | c = delete

### 5.2.2 *Baseline1, Baseline2, Baseline3 and gSpan base analysis after Day6*

The analysis of data by adding the factors like difference between versions of an article, number of edits, and number of disjoint edits has the following results:

==== Detailed Accuracy By Class ====

TP Rate	FP Rate	Precision	Recall	F-Measure	ROC Area	Class
0	0	0	0	0.5	0.5	keep
0	0	0	0	0.5	0.5	merge
1	1	0.641	1	0.781	0.5	delete
Weighted Avg.	0.641	0.641	0.41	0.641	0.5	0.5

=== Confusion Matrix ===

a b c <-- classified as

0 0 20 | a = keep

0 0 3 | b = merge

0 0 41 | c = delete

The above analysis using *LibSVM* shows that adding additional factors actually decreases the accuracy of the result by a huge margin. The above confusion matrix has 20 false negatives for keep and 3 false negatives for merges.

## 5.3 Analysis using Multilayer Perceptron

### 5.3.1 Baseline0 analysis

Analysis using multilayer perceptron has the results as shown below. It shows that it has no improvement over prediction process using *LibSVM*

=== Detailed Accuracy By Class ===

TP Rate	FP Rate	Precision	Recall	F-Measure	ROC Area	Class
0.95	0.045	0.905	0.95	0.927	0.968	keep
1	0	1	1	1	1	merge
0.951	0.043	0.975	0.951	0.963	0.978	delete
Weighted Avg.	0.953	0.042	0.954	0.953	0.953	0.976

=== Confusion Matrix ===

a b c <-- classified as

19 0 1 | a = keep

0 3 0 | b = merge

2 0 39 | c = delete

### 5.3.2 Baseline1, Baseline2, Baseline3 and gSpan base analysis

Baseline1, Baseline2 and Baseline3 do not have much improvement over Baseline 0. The results for *gSpan* baseline are shown in the next page. The confusion matrix shows that there is much accuracy over the number of deletions.

=== Detailed Accuracy By Class ===

TP Rate	FP Rate	Precision	Recall	F-Measure	ROC Area	Class
0.9	0.023	0.947	0.9	0.923	0.981	keep
1	0	1	1	1	1	merge
0.976	0.087	0.952	0.976	0.964	0.987	delete
Weighted Avg.	0.953	0.063	0.953	0.953	0.953	0.986

=== Confusion Matrix ===

```
a b c <-- classified as
18 0 2 | a = keep
0 3 0 | b = merge
1 0 40 | c = delete
```

### 5.3.3 Baseline3 with Augmented Data

This baseline as not discusses in earlier chapters include the data from Day6 of Baseline 3 and also the length of keep, merge and delete. Keep length is the sum of the lengths (words) of description by the users who vote article as keep. Delete length is the sum of the lengths (words) of description by the users who vote article as delete. Merge length is the sum of the lengths (words) of description by the users who vote article as merge

Output analysis for this data using Multilayer Percpetron with Attribute selected classifier is shown below. As you can see it has no improvement over Baseline0. It has 2 false negatives and 1 false positive.

=== Detailed Accuracy By Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	ROC Area	Class
	0.9	0.045	0.9	0.9	0.9	0.99	keep
	0.667	0	1	0.667	0.8	0.995	merge
	0.976	0.087	0.952	0.976	0.964	0.994	delete
Weighted Avg.	0.938	0.07	0.938	0.938	0.936	0.992	

=== Confusion Matrix ===

a b c <-- classified as

18 0 2 | a = keep

1 2 0 | b = merge

1 0 40 | c = delete

## 5.4 Conclusion

The final conclusion can be made that the above input data of analysis does not improve the overall accuracy of prediction by adding factors like difference between versions of an article, number of edits, and number of disjoint edits (according to some contiguity constraints) in addition to the number of votes by much. Multilayer perceptron accurately analysis the input data provided when compared to other classifiers J48 and LibSVM. This is clearly because the above analysis is done over a data of 64 instances. This is clearly a very less sufficient data for analysis. Having a huge amount of data might change the overall analysis by a significant amount.

## 5.5 Future Improvements

In order to improve the overall analysis of results which is the prediction of AfD outcome, we need to gather a considerably huge amount of data. This can be done by crawling through more than one AfD logs from *Wikipedia*. We can also add additional factors which improve the

efficiency of analysis. One such factor is to store the number of citations the user provides when he votes a particular AfD article. By storing the above information there is a high probability of accuracy of prediction process. We can also store information of bunch of *Wikipedia* AfD logs rather than one AfD log.



## References

- Hall, M. (2009). *The WEKA data mining software*. New York: ACM SIGKDD.
- Ian H. Witten, E. F. (2011). *Data Mining Practical Machine Learning Tools and Techniques, Third Edition*. NEW YORK: Morgan Kaufmann publishers .
- Jiawei Han (University of Illinois at Urbana-Champaign), M. K. (2006). *Data Mining: Concepts and Techniques, Second Edition*. New York : Morgan Kaufmann.
- L. Bergroth, H. H. (2000). A Survey of Longest Common Subsequence Algorithms.
- Wikipedia. (2011). *Wikipedia*. Retrieved from [http://en.wikipedia.org/wiki/C4.5\\_algorithm](http://en.wikipedia.org/wiki/C4.5_algorithm)
- Wikipedia. (2011). *Wikipedia*. Retrieved from [http://en.wikipedia.org/wiki/Classification\\_in\\_machine\\_learning](http://en.wikipedia.org/wiki/Classification_in_machine_learning)
- Wikipedia. (2011). *Wikipedia*. Retrieved from [http://en.wikipedia.org/wiki/Category:AfD\\_debates](http://en.wikipedia.org/wiki/Category:AfD_debates)
- Wikipedia. (2011). *Wikipedia*. Retrieved from Wikipedia: [http://en.wikipedia.org/wiki/Wikipedia:AfD\\_categories](http://en.wikipedia.org/wiki/Wikipedia:AfD_categories)
- Wikipedia. (2011). *Wikipedia*. Retrieved from Wikipedia: <http://en.wikipedia.org/wiki/Wikipedia:Categorization>
- Wikipedia. (2011). *Wikipedia*. Retrieved from [http://en.wikipedia.org/wiki/Wikipedia:Polling\\_is\\_not\\_a\\_substitute\\_for\\_discussion](http://en.wikipedia.org/wiki/Wikipedia:Polling_is_not_a_substitute_for_discussion)
- Wikipedia. (2011). *Wikipedia*. Retrieved from [http://en.wikipedia.org/wiki/Wikipedia:Articles\\_for\\_deletion](http://en.wikipedia.org/wiki/Wikipedia:Articles_for_deletion)
- Wikipedia. (2011). *Wikipedia*. Retrieved from [http://en.wikipedia.org/wiki/Wikipedia:Deletion\\_policy](http://en.wikipedia.org/wiki/Wikipedia:Deletion_policy)
- Xifeng Yan, J. H. (2002). gSpan: Graph-Based Substructure Pattern Mining. *gSpan: Graph-Based Substructure Pattern Mining* .
- YAN, X. (2011). *SOFTWARE - gSpan: Frequent Graph Mining Package*. Retrieved from <http://www.cs.ucsb.edu/~xyan/software/gSpan.htm>

## Appendix A - Input Data

Access input data from <http://people.cis.ksu.edu/~a777/>

### A.1 Arff Schema

#### ***Baseline0***

The input format for baseline1 includes the attributes

@relation Base0

@attribute Keep numeric

@attribute Delete numeric

@attribute Merge numeric

@attribute Class { keep, merge, delete }

@data

DATA HERE

#### ***Baseline1***

The input format for baseline1 includes the attributes

@relation DayX

@attribute Difference numeric

@attribute Keep numeric

@attribute Delete numeric

@attribute Merge numeric

@attribute Class { keep, merge, delete }

@data

Data Here

#### ***Baseline2***

The input format for baseline2 includes the attributes

@relation DayX

@attribute Difference numeric

@attribute Number\_of\_nodes\_Difference numeric

@attribute Keep numeric

@attribute Delete numeric

@attribute Merge numeric

@attribute Class { keep, merge, delete }

@data

Data Here

### ***Baseline3***

The input format for baseline3 includes the attributes

@relation DayX

@attribute Difference numeric

@attribute Number\_of\_nodes\_Merged numeric

@attribute Keep numeric

@attribute Delete numeric

@attribute Merge numeric

@attribute Class { keep, merge, delete }

@data

Data Here

### ***gSpan Baseline***

@relation DayX

@attribute Difference numeric

@attribute Frequent\_subgraphOccurence numeric

@attribute Keep numeric

@attribute Delete numeric

@attribute Merge numeric

@attribute Class { keep, merge, delete }

@data

Data Here