M. Tech .(Computer Science) Dissertation Series

DESIGN AND IMPLEMENTATION

OF

CONTENT BASED TEXT RETRIEVAL SYSTEM

A dissertation submitted in partial fulfilment of the requirements for the M . Tech.
(Computer Science) degree of Indian Statistical Institute

By

Nirmalya Barat
(MTC9603)

under the supervision of

Aditya Bagchi
Computer and Statistical Services Centre
INDIAN STATISTICAL INSTITUTE
203 , Barrackpore Trunk Road
Calcutta - 700035

1998

Certificate of Approval

This is to certify that the project work entitled *Design and Implementation of Content based text retrieval system* submitted by Nirmalya Barat , in partial fulfilment of the requirements for M .Tech in Computer Science degree of the Indian Statistical Institute , Calcutta , is an acceptable work for the award of the degree.

Date : July 28 , 1998.

(Supervisor)

(External Examiner)

Acknowledgement

A project work cannot be a one man's job and this project work is not an exception . Above all ,I pay my sincerest gratitude to Prof .Aditya Bagchi , who always believes in the word "we" , for his guidance , advice , enthusiasm and support throughout the course of this dissertation .

I would also like to thank Mr. Subhamoy Maitra , one of my teachers for his valuable suggestions during the course of this project .

My special thank goes to Dr. Bhabatosh Chanda , who played a crucial role in earlier part of my dissertation .

I thank all of my classmates , who gave me numerous suggestions during my project , and also for a friendly atmosphere during my two years at ISI , Calcutta .

Finally , I express my heartiest thanks to the members of the M . Tech . Dissertation Committee.

ABSTRACT

In this project an effort has been taken to implement a content based text storage and retrieval system . The system either can accept the keywords from the user , or the system itself can generate the keywords from an available text and matches them against user queries. Most of the cases the available algorithms have been used with slight modification for ease of implementation . Query refinement has , however , been kept under the supervision of the user instead of leaving it under the control of the implemented retrieval system .

CONTENTS

page No .

I . Introduction 6

II . Content based Text Retrieval 7

III . Design of the System 8

IV . Implementation 11

V . Conclusion 17

VI . Appendix 18

VII . References 22

I . Introduction

As storage is becoming more plentiful and less expensive , the amount of information retained by business and organisations is likely to increase . Searching that information and deriving useful facts, however will become more cumbersome unless new techniques are developed .

We begin with the familiar retrieval situation .We have a collection of documents , any where from hundreds to millions .The documents may be verbal texts, or they may be records of other kinds, such as diagrams or images . From time to time a request will be received to select from the collection , all documents of a specified kind .The *retrieval problem* is to provide for fairly rapid and economical satisfaction of these requests .The problem may become complicated by the frequent addition of new documents to the collection and by the periodic removal of some documents from the collection .

There can be many solutions to the problem .One of them may be , when a request for documents of kind S is received ,read all documents in the collection and select those of kind S. However this is rarely rapid or economical enough at least for human readers .With a suitable machine , of course , retrieval by reading all the documents for every search may be fast enough and may not be too costly . Such system would require a very rapid " reading rate" . However , the documents may be quite large and the work of reading may impose high overhead . We can make this effort simpler by replacing the documents by their short descriptions .

Another approach to the problem may be to divide the documents into small subsets . This will need identification of characteristic properties of all the documents to help in such division . The subsets may not be mutually exclusive . Examples of such properties for division are , *subject heading indexing* , *term indexing* etc . Out of these , term indexing is the most popular one . In this method , the documents are expressed as an unordered conjunction of terms and the request for a kind S of documents is also formulated as an unordered conjunction of terms . Each document which has every term of S is regarded as being of kind S , whatever other terms the document may contain .

A very familiar type of storage organisation for a term indexing is the formation of a subset of collection for each term in the indexing vocabulary . The subset for a term consists of all documents which possess that term .This type of indexing is often called *inverted indexing* . Another type of arrangement consists of document subsets where each document is indexed with all the terms it contains . This indexing method is called *signature indexing* .

II . Content based Text Retrieval

Information retrieval can be of two types *Content based retrieval* or *Concept based retrieval* . In Content based retrieval systems the query contains the specific terms present in the system , however in Concept based retrieval query may contain some terms which are not present in the document collection . These systems mostly use dictionaries , synonyms and homonyms etc to a great extent . This project is based on "Content based retrieval ".

In *content based document retrieval* method documents are retrieved corresponding to a user query which contains the terms present in the required documents . Here we restrict ourselves only to simple text files .Content retrieval for text files is based on locating the files that contain the terms in a user query .The system searches for the collection of all the files of similar structure or content .

The most common application of content based retrieval is the information retrieval from large collection of articles and technical write-ups . Numerous application domains have full-text databases including chemistry , physics, computer science, business magazines etc . Information retrieval systems utilise inverted indexes and other structures to accelerate their searches . In content retrieval based systems , collection of documents are searched and retrieved on the basis of computerised representations . The notion of relevance of a document to a search involves *uncertainty* . For example , if we are looking for documents containing the term "video conferencing" , then it will retrieve documents containing the term "video" or "conferencing" or both .Uncertainty in searching implies that the user posing a query may not get exactly all the documents that satisfy the query . In information retrieval technology two quantities are often used to measure the effectiveness of a search . These are *recall* and *precision* as defined below :

$$\text{Recall} = \frac{\text{Number of relevant Documents returned}}{\text{Total number of relevant Documents in the collection}}$$

$$\text{Precision} = \frac{\text{Number of relevant Documents returned}}{\text{Total number of Documents returned}}$$

Both of these are values in the interval [0,1] . The goal , of course, is to have both recall and precision as close to 1 as possible .

In content based retrieval systems documents are indexed either *manually* or *automatically.* Each index term has local weights in the documents in which it is contained and also has a global weight with respect to the document collection . These are measures giving the relative importance of the term in the corresponding documents and in the collection respectively .The retrieval system defines a *measure* of the documents corresponding to a user query and the documents are arranged in decreasing order of this measure . A predefined number of top ordered documents are returned to the

user . The system also provides means of refining queries in the form of *relevance feedback* . Here the user selects the relevant documents from the returned set of documents and the system generates a modified query by removing the *non-relevant terms* from the query and adding more *relevant terms* to it .The selection of these terms may be guided by the user or the system and it is implementation dependant .

III . Design of the System

Phases :

(1) **System Initialization** - When the user enters the system , it initializes the internal data structures by reading the following system files :

(1 . 1) *stop.dat* - contains pre-existing stopwords .
(1 . 2) *dlenght.dat* - contains the document lengths , this is in fact the number of terms each document contains .
(1 . 3) *headl.dat* - contains the document headlines .
(1 . 4) *docno.dat* - contains the number of documents currently present in the system .
(1 . 5) *doc.dat* - contains the document related information .
(1 . 6) *freq.dat* - containing the term frequency information .

(2) **Text Processing** - Input to this phase is a textual document file .The system provides two types of text processing *automatic* and *interactive* . In automatic mode the system creates a *keyword* (or *index term*) for each *non-stopword* .However in interactive mode the system asks the user before recording any index term . In both the modes the system checks each word of the document against a pre-existing stoplist (stop.dat). If the word is not a stopword , its suffix is stripped off (this process is called *stemming*) resulting a root contained in the original word . This may be noted here that many words may have the same root .This phase also counts the number of occurrences of each stem in the files and records it in the file freq.dat .The number of stems in each file is recorded in dlength.dat file (this will be useful in computing the local weight of the stem in the document file) .This phase is also responsible for generating document id's . The total number of documents present in the system is stored in docno.dat file . The system also takes security measure by providing *password* protection so that unauthorized users cannot enter documents in the system .

File Structures and algorithms used in this phase

(2 . 1) stop.dat : This is a simple text file containing a collection of pre-existing stopwords . A word which is very common in any of the documents present in system , is said to be a stopword . For example the words "and ","the ","for" etc. The system creates an internal data structure in the form of a set of buckets where each bucket is identified by a *hash value*. To allocate a stopword into a bucket we first hash the stopword and then allocate it to the corresponding bucket . Stopwords having the same hash value are arranged in a linked list whose head pointer is stored in the bucket . When checking a word if it stopword the word is at first hashed and then compared with the list of stopwords associated with that bucket .

(2 . 2) freq.dat : The record format of this file is

<stem {document_id stem_freq } > .

Whenever a new record is to be added , an effort is made to check if the stem is already present in the file , if so only a record of the type enclosed within { . . } is added to the corresponding record , otherwise a new record is appended at the end of the file .

(2 . 3) dlength.dat : The record format of the i-th record of this file is

< no_of_indexed_stems_in_ith_document > .

Note that the field here together with the field stem_freq in freq.dat will give the local weight of the stem in the ith_document file .

(2 . 4) docno.dat : This is a simple file containing an integer which gives the number of documents currently present in the system . Each time a file is added to the this number is incremented by 1.This newly generated number can be taken as the id of the newly added document file .

(2 . 5) doc.dat : The record format of this file is

< document_name { stem stem_freq sample_word } > .

Here sample_word is a word of minimum length with the same stem . This sample word is returned at the time of relevance feedback . New records are created at the time of adding new documents to the system .

(3) **Query Processing** : - In this phase the system accepts a query entered by the user . The system provides two types of queries . They may be either *Boolean queries* or *simple queries* (i.e. , non-Boolean) . In case of Boolean queries the system provides the following operators ,

(3 . 1) AND (&) - binary .
(3 . 2) OR (|) - binary .
(3 . 3) NOT (!) - unary .
(3 . 4) DIFFERENCE (~) - binary .

Expressions in a Boolean query can be enclosed by braces ( ) .

At any point of time the user can enter only one type of query . In case of Boolean queries the system checks the validity of the query ; whether some operands are missing or some operators are missing or whether it contains some invalid symbols .

In both types of queries the system checks each word within the query . If it is a stopword (checked through the file stop.dat) or it is not synonymous to any index term or its stem is not contained in the file freq.dat we simply ignore that word in the query . For remaining words in the query the documents name and corresponding stem frequencies are obtained from the file freq.dat .

We define local weight of a stem ti in a document dj as

L(i,j) = freq(ti,dj) / (no . of stems in document dj) .

Again we define the global weight of a stem ti as

G(i) = log (N / Ni) , where log is natural logarithm and

N is the total number of documents currently present in the system out of which Ni documents contain the stem ti .

Next we calculate the measure M(j) for each of the documents dj as

$$M(j) = \sum_{\text{ti is in the query}} L(i,j) * G(j)$$

We arrange the documents in decreasing order of the above measure and top 10 documents are returned to the user .

(4) **Query Modification** : If the above set of returned documents cannot satisfy the user's need, we modify the query by a method called *relevance feedback* . In this phase user selects a set of relevant documents from the above set of documents . The system removes the terms from the query which are present in the non-relevant documents and returns to the user the terms present in the relevant documents . The user then generates a new query by adding terms from the relevant documents . Then this new query is processed again .

(5) **Closing the system** - The system saves the changes made in the internal data structures into the files docno.dat , doc.dat , freq.dat, headl.dat and dlength.dat . This is done if at least one document is entered since the last entry to the system .

IV . Implementation

(1) Porter's Stemming algorithm for suffix stripping

This is an algorithm used for removing suffix of an input word to extract out the stem or root of the word .This process is called *stemming* . For instance the words "begin" , "begins" , "beginning" have the same stem or root namely "begin" . If the word is fully alphabetical then only the stemming is done , otherwise the original word is returned . The algorithm is heavily used in natural language processing , obviously language specific .

The algorithm has three parts :

(1) Part 1 : Check to ensure the word is fully alphabetical .
(2) Part 2 : Run through the Porter's algorithm .
(3) Part 3 : Return the stemmed word , it may be the word itself if word has no suffix or the word is not fully alphabetical .

Detail of the algorithm and the rulelists are given in the **Appendix** .

(2) Algorithm_for_stopword_handling

Initiating stopword list

We use a pre-existing stopword file stop.dat . The file contains a collection of words which are very common in any document . These stopwords are stored in buckets . A hash function accepts a word and generates a hash value to access a bucket . The stopwords having the same hash value are stored in a list whose head pointer is stored within the bucket .

When the system is first entered , the stoplist is initiated by reading the file stop.dat . Here each stopword in the file is first hashed using the hash function :

```
unsigned hash(char *s)
 {
unsigned hashval ;
for(hashval = 0; *s!='\0';s++) hashval = *s + 31 * hashval;
return (hashval % HASHSIZE);
 }
```

Now the bucket with this hash value is tested . If the bucket is empty the pointer of the input stopword is stored within the bucket otherwise the stopword is added at the end of the list whose head node pointer is stored within the bucket .

Checking an input word if it is a stop word

Here we get the hash value and the corresponding bucket from the input word using the same hash function . Now , if the word matches with any of the stopword in the list associated with that bucket then the input word is a stopword otherwise not .

(3) Algorithm for synonym handling

The user query may contain some terms which are not present in the system . However there can be index terms synonymous to these terms .So while searching for those query terms , we should be able to replace those terms by corresponding index terms .

Here we use a pre-existing synonym file syn.dat where each record is of the format < root , {key} >, where root is an indexed term and key is its one of the synonyms . The synonyms are stored in a synonym table with the record format <root , key> . There can be several entries of the form shown above for a fixed root value . synonym_table is sorted with respect to the key field .

Initiating synonym table

When the user enters the system , the system reads the syn.dat file if it exists .Then it reads records from this file and creates the synonym table entries of the above form . If the file syn.dat does not exist synonym handling is bypassed .

Looking up synonyms in synonym table

Input to this algorithm is a query term . If the input term is an index term the original word is returned . Otherwise we search the synonym table until we get a record of the form < root , key > where the key matches with the input query term .If a record is obtained the algorithm returns the corresponding root . Otherwise if such record cannot be obtained algorithm returns the original word .

(4) Algorithm for document length handling

The system records document length in a special way .It counts the number of index terms (i.e. , stems) when a file is first entered into the system .This count is taken to be the size of thedocument .The document lengths are stored into a file dlength.dat where the records are of the form < document_length > .The i-th record of the file contains the length of the document with id i .

The document lengths are stored internally in a dynamic array *doc_length_array* .When the user enters the system the doc_length_array is initiated by reading the file dlength.dat .When any document file is added to the system its length is computed and is added to doc_length_array . It stores the length of the file with document id "docid" in

doc_length_array[docid] .The size of the doc_length_array is the number of documents currently present in the system . This number can be obtained from docno.dat file . During exit the system saves the doc_length_array content in the file dlength.dat .

Algorithm for getting the document length given its id

It simply returns doc_length_array[id] .

(5) Algorithm for document headline handling

The document headline is entered when a document is entered into the system . It can be atmost 250 characters long .The document headlines are stored into a file headl.dat with the records of the form < document_headline > . The i-th record of the file contains the headline of the document with id i . If the headline of a document is not specified it is taken to be NULL . Each document contains its headline in the first line which is preceded by a # sign identifying it as the document headline . However if such a line is not present , the document headline is taken to be NULL .

The document headlines are stored internally in a dynamic array *headline_array* . When the user enters the system the headline_array is initiated by reading the file headl.dat .

It stores the headline of the file with document id "docid" in headline_array[docid] . The size of the headline_array is the number of documents currently present in the system . This number can be obtained from docno.dat file . During exit , the system saves the headline_array content in the file headl.dat .

Algorithm for getting the document headline given its id

It simply returns headline_array[id] .

(6) Text Processing algorithm

This algorithm is followed when a new document is added to the system . The input to this algorithm is a textual document file name . This algorithm indexes the document to find the terms which can be used to find the documents corresponding to a user query .

Text processing can be either automatic or interactive . Before starting the processing the system first confirms whether the mode is automatic or interactive . In automatic mode the system records all non-stopwords present in the document as index terms , while in the interactive mode the system asks the user before adding any non-stopword . Though interactive mode is a bit time consuming , it is often preferable because index file's size can be quite large if the stopword file is not good in case of automatic mode . It may index some terms which are not so important .

When a user enters a document file name , the system first checks whether the file already exists in its directory .If the file is already added to the system which is checked through doc.dat file , processing request is ignored .

On the contrary , the system starts actual processing of the text file .It first checks whether the first line of the file contains a # symbol , if so , it is taken as the file's headline , else NULL is taken as the headline of the file .Now the system reads the number of documents currently present in the system from docno.dat file for generating new document id .

The system uses a *binary tree* for storing the index terms contained in the file . Here each node consists of the stem name representing the term, stem frequency in that file and a minimum length sample word with the same stem which will be useful in relevance feedback .

The system reads one line of text at a time from the file and breaks it into the constituent words . For each word it checks whether or not it is a stopword through the stop.dat file . If the word is not a stopword it is first stemmed and then added to the binary tree mentioned above and node frequency is adjusted as well .The step above is repeated until all lines are over .

After this , if the file contains at least one index term a new document id is generated , headline is added to the headline array .The binary tree is traversed inorderly , the term frequency information are added to the term bucket , document information are added to doc_array , and total number of terms in the file is stored in the doc_length_array .

(7) Query Processing Algorithm

Query processing starts when user selects the "Enter Query" option in the main menu . The system accepts a query entered by the user . Queries can be either simple or Boolean . User can enter only one type of query at a time and it is to be specified by the user in advance . Query processing is done if the system contains at least one document file .

Processing of simple queries

The query processing is initiated only if there is at least one document in the system . The system breaks the user query into constituent words . For each word , the system first checks whether it is a stopword or not . If it is so the word is ignored , else the word is stemmed and

the stem is again checked against stopword list .When an index term is present in all the documents , its global weight is zero (according to our definition of global weights). The term is temporarily stored into the internal stopword collection to bypass searching of the term . However , if the stem is not a stopword it is used for searching again .

For storing the search information the system uses a dynamic array named *simp_doc_score_array* having one entry for each of the documents obtained during the search . The array elements are of the following format :

```
int doc_id ;
double score ;
int no_of_terms_covered ;
```

While searching for a relevant term the system first searches the term in the term bucket. If it gets the corresponding record there , it computes term's local weight in each of the documents containing the term and also the term's global weight in the system (formulas are given in section III) . After this , the contribution of the term in the scores of the documents are computed by taking the product of the global weight with the corresponding local weights and are added to the score field of the corresponding entries in the simp_doc_score_array .

After searching all the relevant terms present in the query the system sorts the simp_doc_score_array with respect to the score field in descending order and a pre-specified number of top ordered documents in this list are returned to the user as the result of the query processing .

Processing of Boolean Queries

In case of Boolean queries the system provides the following operators

(3 . 1) AND (&) - binary .
(3 . 2) OR (|) - binary .
(3 . 3) NOT (!) - unary .
(3 . 4) DIFFERENCE (~) - binary .

Expressions in a Boolean query can be enclosed by braces ( ) .

When the user enters the query the system checks whether the query contains only valid symbols . The query is also checked for balanced parentheses . If the above check is true the *postfix form* of the above query is obtained . Here the postfix query string contains all the tokens separated by blank spaces .

Next step in processing is the evaluation of this postfix query . The system uses a dynamic array called *search_result_array* which contains an entry for each of the terms in the query or result of some Boolean operations on these terms . The entries are of the following structure,

```
int status , this is 0 if the term is a stopword
                    1 if the term is an index term
                    2 if the term is not present in the system
char *stem , pointer to the term
```

long operand_id , this is the index position of the entry in the array
long nDocs , the number of documents containing the term

it also contains a pointer to an array containing document_id and score entry for each document containing the term .

For processing the query the system uses a stack for storing the operand_id of the terms present in the system or results of Boolean operations on these terms .

For each token in the postfix query if it is a term it is searched by the system in the term bucket and an entry is made in the search_result_array . The operand_id ,i . e . , the index position in the search_result_array is saved onto the stack or else if it is a binary operator symbol , the top two entries on the stack are popped out and Boolean operation is performed on the search_result_array entries given by those two operand_ids and result is stored in one of the entries and its operand_id is saved onto the stack . If the operator is unary , only one element is popped out and the operation is performed on the corresponding entry . Popping from an empty stack indicates missing operand error .

While making an entry in the search_result_array if the word or its stem is a stopword an entry is made with status 0 , nDocs 0 and score 0 . If the term is not present in the system a similar entry is made with status 2 . However , if the term is an index term an entry is made with relevant information as in the case of simple queries .

At the end of processing , the result_operand_id is left onto the stack and a pre-specified number of top scored documents are returned to the user after sorting the array in descending order with respect to the score field .

The Boolean operations on the search_result_array elements are defined as follows .Let set1 and set2 be two search_result_array elements and op be a Boolean operation as mentioned above . Let result be the resultant search_result_array element . Then ,

(a) if op = OR (|) , then the array pointed to by the result will contain all the document ids and corresponding scores which are present in at least one of the arrays pointed to by set1 and set2 . For any document_id which is present in both the arrays its scores are added to get the score in the result .

(b) if op = AND (&) , then the array pointed to by the result will contain all the document ids and corresponding scores which are present in both the arrays pointed to by set1 and set2 . For any document_id which is present in both the arrays its scores are added to get its score in the result . In case of AND operator if at least one of terms corresponding to set1 or set2 is a stopword then the AND operator is replaced by OR operator only to ignore the stopword . For example if A is a stopword and B is an index word then result of A & B would have resulted into no documents , as , A is not an index term , though by the definition of stopwords they occur in all the documents .

(c) if op = DIFFERENCE (~) , then the array pointed to by the result will contain all the document ids and corresponding scores which are present in the array pointed to by set1 and but not in the array pointed to by set2 . For any document_id which is present in the array pointed to by set1 its score is retained in the result .

(d) if op = NOT (!) , then the array pointed to by the result will contain all the document ids and corresponding scores which are not present in the array pointed to by set1 . However , unlike the other cases , here any document is awarded a zero score for not containing any term .

Relevance feedback

If the above set of returned documents cannot satisfy the user's need, we modify the query by a method called relevance feedback . In this phase user selects a set of relevant documents from the above set of documents . The system uses a *relevance_array* which is constructed at the time of returning the documents to the user (if any) . The elements are of following structure :

```
int serial_no ; /* serial_number of the document in the result displayed */
int not_relevant ; /* 0 if relevant */
long doc_id ;
```

When the user selects "Relevance Feedback" option in query menu system gets the user's feedback to set the not_relevant flag of the entries of the relevance_array . Now the system constructs a relevant term list by adding all the terms present in the relevant documents removing all the terms present in the non-relevant documents and returns the list to the user . The user then generates a new query by selecting terms from this relevant list . Then this new query is processed again in the usual manner .

V. Conclusion

This M.Tech. dissertation deals with the implementation of a content based text retrieval system . Since known algorithms with slight modifications have been used , quality or limitation of the work is restricted by the quality or limitations of the algorithms used . However, for query modification, relevance feedback method could not be implemented for want of time . This may be considered as an immediate future work if the system has to be used for any application. Integration of text retrieval and database is an area of topical research . The outcome of the current effort may be taken as a component to such future work .

VI . Appendix

Details of the Porter's stemming algorithm and the rulelists used

Here the algorithm applies a set of rewrite rules on the input word to determine the largest suffix of the word to be replaced . Each of these rules are in fact set of subrules . Here the algorithm loops through the rule sets until a match meeting all conditions is found . If a rule fires it return its id , or 0 if no rule is fired . Conditions on the length of the root are checked as a part of this phase .

The rulelists are of the following structure

```
int id;                 /* returned if rule fired */
char *old_end;          /* suffix to be replaced */
char *new_end;          /* suffix to be replaced with */
int old_offset;         /* from end of word to start of suffix */
int new_offset;         /* from beginning to end of new suffix */
int min_root_size;      /* min root word size for replacement */
int (*condition)();     /* the replacement test function */
```

In accordance with the above structure the rulelists are as follows . Here LAMBDA stands for empty string .

(i) step1a_rules

```
101, "sses" , "ss"        , 3,  1, -1, NULL,
102, "ies"  , "i"         , 2,  0, -1, NULL,
103, "ss"   , "ss"        , 1,  1, -1, NULL,
104, "s"    , LAMBDA      , 0, -1, -1, NULL,
000, NULL, NULL           , 0,  0,  0, NULL,
```

(ii) step1b_rules

```
105, "eed"  , "ee"        , 2,  1,  0, NULL          ,
106, "ed"   , LAMBDA      , 1, -1, -1, ContainsVowel,
107, "ing"  , LAMBDA      , 2, -1, -1, ContainsVowel,
000, NULL   , NULL        , 0,  0,  0, NULL          ,
```

(iii) step1b1_rules

```
108, "at",   "ate" ,   1, 2, -1, NULL,
109, "bl",   "ble" ,   1, 2, -1, NULL,
110, "iz",   "ize" ,   1, 2, -1, NULL,
111, "bb",   "b"   ,   1, 0, -1, NULL,
112, "dd",   "d"   ,   1, 0, -1, NULL,
113, "ff",   "f"   ,   1, 0, -1, NULL,
```

```
114, "gg",   "g"  ,    1, 0, -1, NULL,
115, "mm", "m" ,    1, 0, -1, NULL,
116, "nn"        , "n"      ,    1, 0, -1, NULL,
117, "pp"        , "p"      ,    1, 0, -1, NULL,
118, "rr"        , "r"      ,    1, 0, -1, NULL,
119, "tt"        , "t"      ,    1, 0, -1, NULL,
120, "ww"        , "w"      ,    1, 0, -1, NULL,
121, "xx"        , "x"      ,    1, 0, -1, NULL,
122, LAMBDA , "e"      ,   -1, 0, -1, AddAnE,
000, NULL       , NULL ,   0, 0, 0, NULL,
```

(iv) step1c_rules

```
123, "y"      , "i"      ,    0, 0, -1, ContainsVowel,
000, NULL , NULL ,    0, 0, 0, NULL,
```

(v) step2_rules

```
203, "ational" ,  "ate"  ,   6, 2, 0, NULL,
204, "tional"  ,  "tion" ,   5, 3, 0, NULL,
205, "enci"    ,  "ence",    3, 3, 0, NULL,
206, "anci"    ,  "ance",    3, 3, 0, NULL,
207, "izer"    ,  "ize"  ,   3, 2, 0, NULL,
208, "abli"    ,  "able",    3, 3, 0, NULL,
209, "alli"    ,  "al"   ,   3, 1, 0, NULL,
210, "entli"   ,  "ent"  ,   4, 2, 0, NULL,
211, "eli"     ,  "e"    ,   2, 0, 0, NULL,
213, "ousli"   ,  "ous"  ,   4, 2, 0, NULL,
214, "ization" ,  "ize"  ,   6, 2, 0, NULL,
215, "ation"   ,  "ate"  ,   4, 2, 0, NULL,
216, "ator"    ,  "ate"  ,   3, 2, 0, NULL,
217, "alism"   , "al"    ,   4, 1, 0, NULL,
218, "iveness" , "ive"   ,   6, 2, 0, NULL,
219, "fulnes"  , "ful"   ,   5, 2, 0, NULL,
220, "ousness" , "ous"   ,   6, 2, 0, NULL,
221, "aliti"   , "al"    ,   4, 1, 0, NULL,
222, "iviti"   , "ive"   ,   4, 2, 0, NULL,
223, "biliti"  , "ble"   ,   5, 2, 0, NULL,
000, NULL      , NULL,    0, 0, 0, NULL,
```

(vi) step3_rules

```
301, "icate" ,  "ic"       ,   4, 1, 0, NULL,
302, "ative" ,  LAMBDA ,   4, -1, 0, NULL,
303, "alize" ,  "al"       ,   4, 1, 0, NULL,
304, "iciti" ,  "ic"       ,   4, 1, 0, NULL,
```

```
305, "ical"    , "ic"          ,  3, 1, 0, NULL,
308, "ful"     , LAMBDA , 2, -1, 0, NULL,
309, "ness" , LAMBDA ,  3, -1, 0, NULL,
000, NULL , NULL     ,  0, 0, 0, NULL,
```

(vii) step4_rules

```
401, "al"    ,   LAMBDA , 1, -1, 1, NULL,
402, "ance" ,   LAMBDA, 3, -1, 1, NULL,
403, "ence" ,   LAMBDA, 3, -1, 1, NULL,
405, "er"    ,   LAMBDA, 1, -1, 1, NULL,
406, "ic"    ,   LAMBDA, 1, -1, 1, NULL,
407, "able" ,   LAMBDA, 3, -1, 1, NULL,
408, "ible" ,   LAMBDA, 3, -1, 1, NULL,
409, "ant"    ,   LAMBDA, 2, -1, 1, NULL,
410, "ement",   LAMBDA, 4, -1, 1, NULL,
411, "ment" ,   LAMBDA, 3, -1, 1, NULL,
412, "ent"    ,   LAMBDA, 2, -1, 1, NULL,
423, "sion"   ,   "s"         , 3, 0, 1, NULL,
424, "tion"   ,   "t"         , 3, 0, 1, NULL,
415, "ou"     ,   LAMBDA , 1, -1, 1, NULL,
416, "ism"    ,   LAMBDA , 2, -1, 1, NULL,
417, "ate"    ,   LAMBDA , 2, -1, 1, NULL,
418, "iti"    ,   LAMBDA, 2, -1, 1, NULL,
419, "ous"    ,   LAMBDA, 2, -1, 1, NULL,
420, "ive"    ,   LAMBDA, 2, -1, 1, NULL,
421, "ize"    ,   LAMBDA, 2, -1, 1, NULL,
000, NULL  ,   NULL    , 0, 0, 0, NULL,
```

(viii) step5a_rules

```
501, "e"     ,   LAMBDA , 0, -1, 1, NULL,
502, "e"     ,   LAMBDA , 0, -1, -1, RemoveAnE,
000, NULL ,   NULL     , 0, 0, 0, NULL,
```

(ix) step5b_rules

```
503, "ll"     ,     "l",    1, 0, 1, NULL,
000 , NULL , NULL ,    0, 0, 0, NULL,
```

The algorithm applies the rules (i) through (ix) and gets the rule id if one is fired . However , if rule id is either 106 or 107 then only step1b1_rules are tested .

For the application of the rules , the input word is scanned from the end and the largest suffix of the word that matches with old_end field of a rule is replaced by its *new_end* field only if the *minimum_root_size* and the condition field of the rule are satisfied . The condition fields can have the following functional values :

(i) *NULL* ,i . e . , no condition is actually required .

(ii) *ContainsVowel* :obviously, under the definition of a vowel, a word contains a vowel iff either its first letter is one of "aeiou", or any of its other letters are "aeiouy" . The plan is to test this condition . This function returns TRUE if the word contains a vowel else it returns FALSE .

(iii) *AddAnE* : Function returns TRUE if the current word meets special conditions for adding an e . We Check for size of 1 and a *consonant-vowel-consonant* ending , i.e. , if the current word ends with a consonant-vowel-consonant combination, and the second consonant is not w, x, or y, FALSE otherwise . We look at the last three characters .

(iv) *RemoveAnE* : Rule 502 applies only to a root with this characteristic . We check for word size of 1 and no consonant-vowel-consonant ending .

The WordSize used here is computed as follows :

We count syllables in a special way: count the number vowel-consonant pairs in a word, disregarding initial consonants and final vowels . The letter "y" counts as a consonant at the beginning of a word and when it has a vowel in front of it; otherwise (when it follows a consonant) it is treated as a vowel .

We run a DFA to compute the word size . The initial state 0 checks the first letter . If it is a vowel, then the machine changes to state 1, which is the "last letter was a vowel" state . If the first letter is a consonant or y, then it changes to state 2, the *"last letter was a consonant state"* . In state 1, a y is treated as a consonant (since it follows a vowel), but in state 2, y is treated as a vowel (since it follows a consonant . The result counter is incremented on the transition from state 1 to state 2, since this transition only occurs after a vowel-consonant pair, which is what we are counting .

VII . References

(1) Mandar Mitra , Information Retrieval , seminar lecture at ISI Calcutta (1997) .

(2) T. Letsche and M.W. Berry , Large-Scale Information Retrieval with Latent Semantic Indexing , Information Sciences 100 , 105 - 137 (1997).

(3) Setrag Khoshafian and A. Brad Baker , Multimedia and Imaging Databases , Morgan Kaufmann (1996).

(4) J. H. Shera , A. Kent and J. W. Perry , Information Systems in Documentation , volume 2 , Interscience publishers , Inc. , New York (1957).

(5) J. H. Shera , Information Retrieval and Machine Translation , volumeIII ,part1, Interscience publishers , Inc. , New York (1960).