The Design and Optimisation of Stemming Algorithms

Rob Hooper

r.p.hooper@lancaster.ac.uk

Abstract:

The performance of information retrieval systems can be improved by matching key terms to any morphological variants. A suffix removal stemming algorithm is a procedure for automatically conflating semantically related terms together via the use of rule sets. The project implements a method for evaluating stemming algorithms and measures the performance of the main three stemmers, Lovins, Paice and Porter. Information about stemmer behaviour is retained during operation of the algorithms and is processed to obtain a series of possible enhancements to the Lovins rule sets. These optimisations are implemented and tested using the above evaluation method and clear improvements over the original algorithm are demonstrated.

1) Introduction:

1.1) Context:

In the context of modern computing information retrieval has become focussed on the extraction of key terms from documents to aid in the accessing and navigation of web pages and electronic information [17]. These key terms are used in the process of information retrieval to obtain relevant answers to questions by interrogating a database of documents [15]. The two main questions are;

- Should a document be retrieved?
- What level of relevance is the document to the user?

To answer these questions the system must take input from the user in the form of key terms, as mentioned above, and attempt to link these terms supplied to the information contained within the documents.

The above technique of retrieval leads to the creation of a new question of how to decide whether a document and a query have a particular key term in common. The obvious solution is to simply test for absolute equality between the key term and all terms in the document and retrieve it if a match is found. However the key terms may have many morphological variants that share similar semantic interpretations and it may be beneficial for the information retrieval system to consider these terms equivalent. In order to recognise these variants the system will require some form of natural language processing. One form of natural language processing that can be used to achieve recognition is an algorithm that will strip a term to its morphological root via the removal of prefixes and suffices. [10]

The term *conflation* is used to describe the process of matching a term to its morphological variants. There are many possible ways of performing *conflation*. These include manually entering extra information in the search query, either in the form of endings that can be concatenated to the key terms when searching or by manually listing all variants that the user wishes to be considered. A more common way of integrating this matching into a retrieval system is to automate the process via the use of *stemmers* [3]. Stemming algorithms will attempt to take a word and remove suffices from it, thus reducing it to its morphological *root* or *stem*. The decisions on which endings should be removed and in what circumstances they should be removed are made using ending and/or rule lists that are searched during the *stemming* process.

This process presents the information retrieval system with alternative methods for matching morphological variants to the key terms. The first method involves performing the stemming of the document in advance, to produce a list of stems that exist within the document. This will require less storage than a summary of all relevant terms and also will reduce the computational load during searching of a document. However it is not possible to draw on any extra information in relation to *weighting* a terms relevance. If the stemming is performed during retrieval then it will be possible to *weight* the terms so preference is given to exact matches of the key term, compared to matches of morphological variants. This advantage is gained at the expense of retrieval efficiency and the final decision will remain subjective. [5]

1.2) Statement of Aim:

The following section will list the aims of the project, each aim will be accompanied with an explanation of the rationale behind it and also the methodology that will be used in achieving each aim. The aims will be split into two categories, primary and secondary, with the secondary aims achievement being dependent upon timeliness constraints dictated by the completion of the primary goals.

Primary Aims:

- Demonstration of correctness: Testing has been performed on the tool but currently no conclusive demonstration of the calculation programs correctness has been performed. This step will involve the creation of hand calculated sets of results for three different grouped word sets and then using these to verify the correctness of the program.
- Optimise and refine the current evaluations tool, including;
  - Update GUI: The GUI is in need of updating to utilise either the Swing or SWT graphics API, this will improve the usability of the program.
  - Improve efficiency: Currently the sorting and calculation algorithms lead to unacceptable delays when performing the calculations. One step to be taken will include storing

information about the start points in the list of words beginning with certain two letter combinations as the first two letters of a term never change when performing stemming. By combining this information into a more efficient sorting algorithm the tool will be able to assist in rapid changing and testing of new rule sets.

- Debugging: Two major bugs are currently known and include the inability to resume the sorting a word file from a previous session and the non-deletion of temporary files that lead to a system crash when more than one test is run without restarting.

- Implement additional Stemming Algorithms: At the moment only the Lovins algorithm has been implemented into the program and additional algorithms need to be implemented to provide comparative results and to gather extra information. This will involve developing an interface, called *Stemmer*, that will allow new algorithms conforming to its definition to be added easily to the program. By modifying existing implementations of the Paice and the Porter stemmers the extra functionality can be added to the system.
- Gather information regarding the steps taken by each algorithm: The basis for optimising the stemmer rule sets will be the retention of information regarding a 'stemmer's history', the steps taken in reducing a word to its stem. The current implementation of the Lovins stemmer retains the following information;
  - Original term and Stemmed term,
  - Ending(s) removed,
  - Exceptions fired,
  - Recodes Performed,

  From this information statistics regarding the frequency of use and errors can be derived. By ensuring this information is also retained in the new implementations of the Paice and Porter algorithms, such data can be gathered for all available grouped word sets.
- Use information collected to attempt to optimise the rule sets for the Lovins algorithm: The examination of the information gathered above will allow reoccurring errors within algorithms to be identified and removed/modified to increase the accuracy of the stemmer. This will be performed systematically with tests being run to show the effects of any changes and whether additional errors are introduced that negate the advantages of the change. The decision has been made to focus on the Lovins stemmer as previous work [16] & [5] have shown it to offer the greatest scope for improvement

Secondary Aims:

- Attempt an optimisation of the rule sets for the Paice and/or Porter algorithm: Time allowing extra information gathered from the additional stemmers implemented will allow for the optimisation of the corresponding rule sets and also may prove useful in correcting errors in the Lovins rule set.
- Develop more and larger grouped word sets: The need for these rule sets will become evident

once the initial optimisation begins. With the largest grouped word set currently c.10,000 the validity of results could be questionable, and also significance tests on changes to the rule sets are unlikely to be conclusive. The amount of time available will dictate the number and/or size of the additional word sets produced

The overall aim of the project is to build on the previous work [5] and attempt to verify the original findings by examining the results more thoroughly. Once the verification and updates have been performed the project aims to produce a systematic process for optimising and refining the rule sets of stemming algorithms, which will include automatically calculated performance statistics for endings/rules.

2) Background Information:

2.1) Background to Stemming:

Morphology is the area of linguistics that is concerned with the internal structure of words. There are two subclasses of morphology, *derivational* and *inflectional*. It is the second of these classes that is of interest when attempting to reduce a word to its morphological *root* or *stem*. [9] *Inflexion* is formally defined as,

"Inflexion is the process by which words (principally nouns, verbs, adjectives and adverbs) changes their form, especially their ending, in accordance with their grammatical role in a sentence" [18]

Some of the basic rules of inflexion involve the plural and possessive forms of nouns and the past and progressive form of verbs. Conflation is concerned with attempting to 'reverse' the inflexion process by performing the inverse operation related to the basic inflexion rules. There has been much research into the improvements that can be made to information retrieval systems by matching variants of words. An experiment in the Hebrew language showed that,

"A failure to process morphological variants results in retrieving only 2%-10% of the documents retrieved with such processing"

These results infer that a similar process applied to English, although a weaker morphological language than Hebrew, could improve the performance of information retrieval systems. Many of the processes available for matching terms are displayed in the following taxonomy provided in [3];

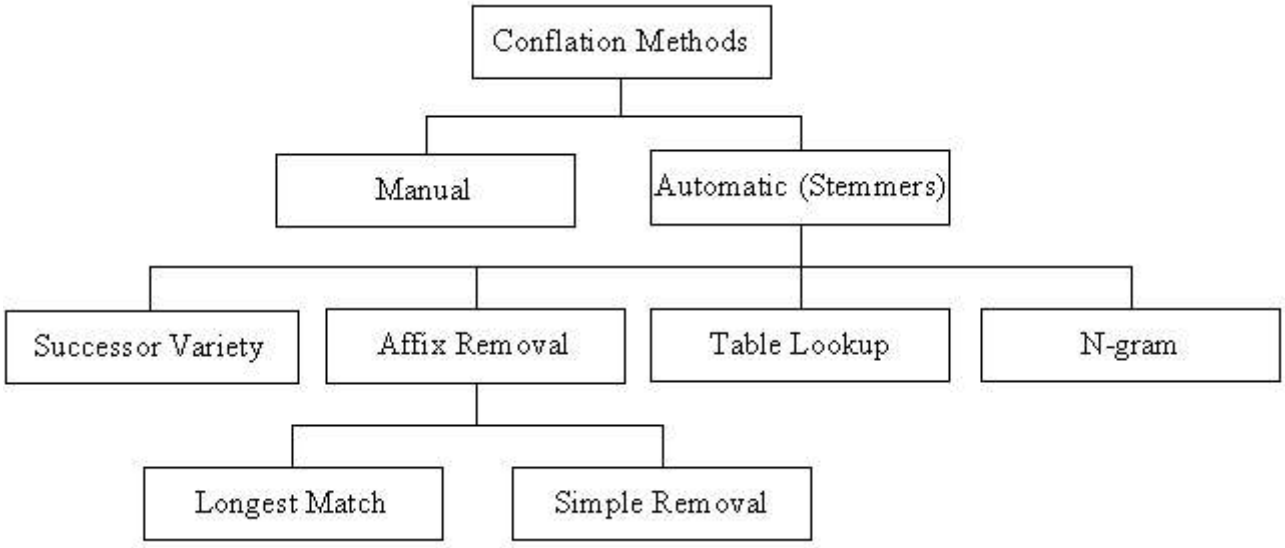

Figure 2.1 Conflation Methods [3]

Other techniques are detailed in [10] that include strict truncation, word segmentation, linguistic morphology, suffix removal and letter bigrams.

There are inherent problems that exist when attempting to conflate English words, one of the more difficult problems is the existence of *strong verbs* that follow no set pattern for inflexion and will change their stem when forming tenses, e.g. *throw, threw, thrown*. Other verbs are completely irregular, e.g. *go, went, gone*.[18] These non-formulaic changes are unpredictable and conflation without the use of a lexicon is virtually impossible without introducing errors. This problem is highlighted in [9] as one of the main causes of errors in stemming algorithms. Krovetz [9] proposes that the problem with conflation methods is that they operate without a lexicon and thus ignore word meaning, which leads to a number of stemming errors, both with words that are unrelated being conflated together and unrelated terms being matched.

However the methods proposed to tackle this issue are complex and to maintain a efficient and effective approach to conflation stemming algorithms have been developed that accept that some errors will occur, but the trade off is made with improved performance.

2.2) Stemming Algorithms:

Affix removal conflation techniques are referred to as *stemming algorithms* and can be implemented in a variety of different methods. All remove suffices and/or prefixes in an attempt to reduce a word to its stem [3]. The algorithms that are discussed in the following sections, and those that will be implemented in this project, are all suffix removal *stemmers*.

During the development of a stemmer the issues of iteration and context awareness must be addressed. Suffices that are concatenated to words are often done so in a certain order, such that a set of *order-classes* will exist among suffices. [5] An iterative stemming algorithm will remove suffices one at a time, starting at the end of the word and working towards the beginning. [11]

An issue also exists about whether a stemmer should be *context-free* or *context-sensitive*. A context-sensitive algorithm involves a number of qualitative contextual restrictions that are developed to prevent the removal of endings that, in certain situations, can lead to erroneous stems being produced. A context free algorithm removes endings with no restrictions placed on the circumstances of the removal. [11]

The following sections will describe the various stemming algorithms that have been developed since the publication of the first algorithm in 1968.

2.2.1) Lovins:

The Lovins stemming algorithm was first presented in 1968 [11] by Julie Beth Lovins. It is a single pass, context sensitive stemmer, which removes endings based on the *longest-match* principle. The stemmer was the first to be published and was extremely well developed considering the date of it's release and has been the main influence on a large amount of the future work in the area [W2] The algorithm has been criticised for being designed for use in both information retrieval systems and linguistics, which has led to reasonable results in both areas, but an inability to excel in either. [5] It has performed poorly in recent evaluations of stemming algorithms constantly falling below the performance of the more common Porter and more recent Paice/Husk algorithms [5], [7], [13] & [16]

The stemmer is considered complex for its age and utilises a large list (297) of endings that can be removed and also a list of exceptions and recodes. The stemmer utilises this large ending list in an attempt to avoid an iterative stemming process, whilst still removing more than one *order-class* of endings [11]. The list contains all common word suffices, such as *ing* and *ly*, and also valid combinations, *ingly*, of these endings, thus no iterative step is required. The endings are sorted firstly by length and then alphabetically to improve efficiency. [11] By having the endings sorted by length the first matching ending found will be the longest, this negating the need for further searching as the stemmer is based on the longest-match principle. The efficiency is also improved by ascertaining a *start point* within the list for each word. This start point is determined by the length of the word to prevent attempting to match endings to words which are shorter than it.

As mentioned above the stemmer is context-sensitive and each ending is associated with one of a number of qualitative contextual restrictions that prevent the removal of endings in certain circumstances. The stemmer utilises a number of rules that are designed to cope with the most common exceptions. All endings are associated with the default exception, that a stem must be at least two letters long, which is designed to prevent the production of ambiguous stems. Other rules assert one of the following conditions on the ending's removal,

- Increasing the minimum length of a stem following a ending's removal.
- Preventing the removal of endings when certain letters are present in the remaining stem.
- Combinations of the above restrictions. [11]

When developing the stemmer Lovins [11] described the most desirable form of context sensitive rule as one that can be generalised to apply in numerous situations. During the development of the stemmer it was discovered that few examples of such rules could be found. For each ending a number of special cases exist that cause erroneous stems to be produced, these are often unique to the ending and a number of rules would have to be developed that would prevent only a small number of errors. This process would require large amounts of time and data, and would offer

diminishing improvements in performance over time. For this reason it was decided to deal with the more obvious exceptions and to hopefully limit the number of errors that remain unaccounted for in the exception list. [11]

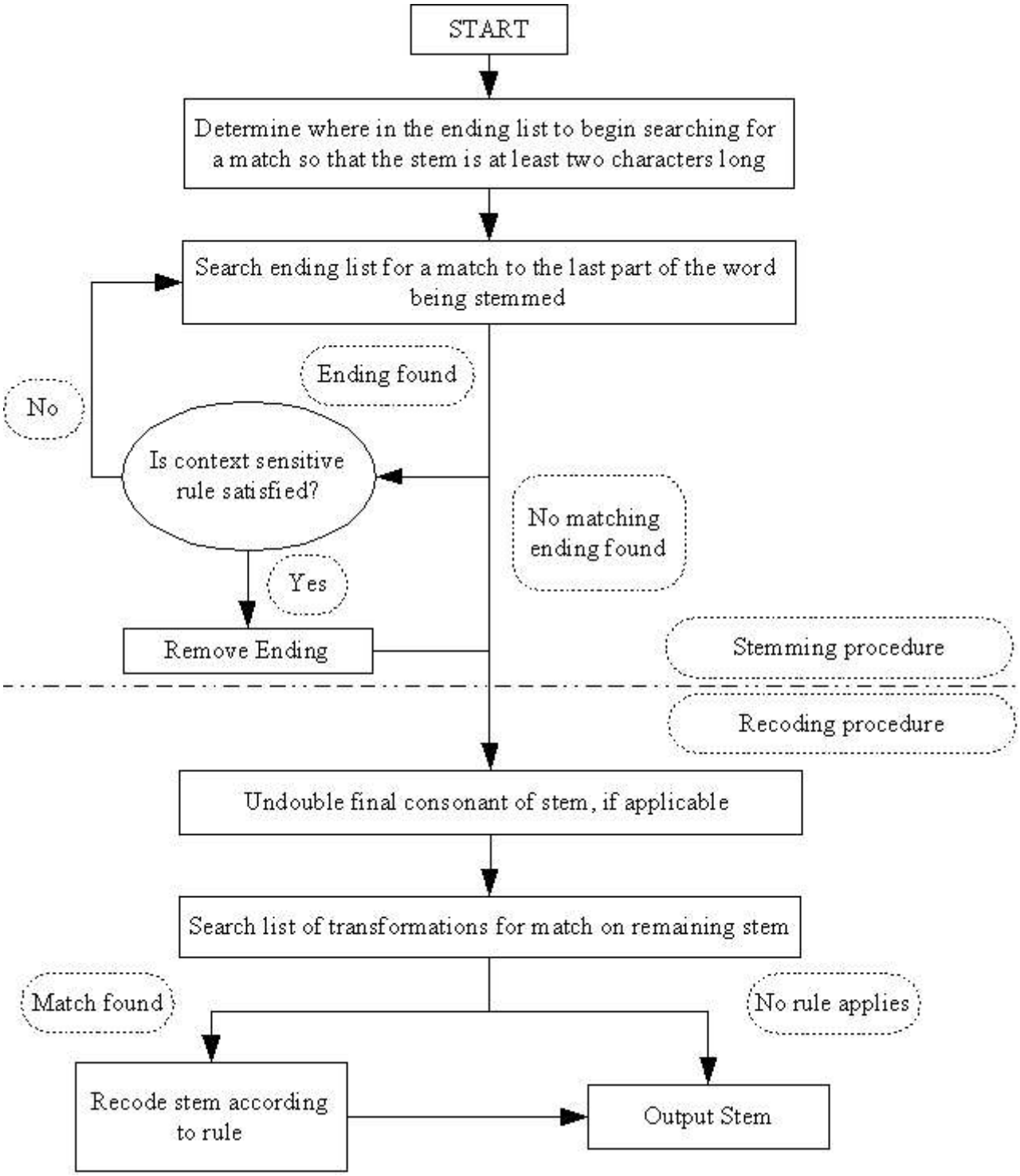

Figure 2.2 Lovins Stemmer [11]

The flowchart shown above details the complete algorithm proposed by Lovins [11] and illustrates the two main stages of the algorithm. The *stemming phase* has been discussed above and includes the removal of endings and the testing of associated exceptions among other steps. The second part of the algorithm is the *recoding phase*.

The term *spelling exception* is used to cover all the circumstances in which a stem may be spelled in more than one way. The majority of these exceptions that occur in English are due to "Latinate derivations" [11], such as *matrix* and *matrices*. Other types of exceptions occur that can be attributed to differences in British and American spellings, such as *analysed* and *analyzed*, or to basic inflexion rules that cause the doubling of certain consonants when a suffix is added. [11]

Two ways are proposed by Lovins to deal with this problem which are called *recoding* and *partial matching*. She defines them as follows,

"*Recoding occurs immediately following the removal of an ending and makes such changes at the end of the resultant stem as are necessary to allow the ultimate matching of varying stems"* [11]

"Partial matching operates on the output from the stemming routine at the point where the stems derived from catalogue terms are being searched for matches. All 'partial' matches, within certain limits, are retrieved rather than just all 'perfect' matches" [11]

From the quotes it can be seen that the main difference between the two techniques for dealing with spelling exceptions is that recoding can be considered part of the stemming algorithm whereas partial matching cannot. The relevant merits and problems associated with both methods are discussed in [11] and it was decided that recoding would be used as the knowledge available during the development of the stemmer allowed Lovins to argue that although partial-matching is likely to lead to more correct matches, it was felt these benefits would not be sufficient to merit the increased complexity, and subsequent loss of efficiency, the partial-matching technique causes. For these reasons recoding is used within the Lovins stemmer via a series of steps illustrated within figure 2.2.

2.2.2) Paice/Husk:

The Paice/Husk stemmer was first published in 1990 [14] and was developed by Chris Paice with the assistance of Gareth Husk. The stemmer is a conflation based iterative stemmer. The stemmer, although remaining efficient and easily implemented, is known to be very *strong* and aggressive [4]

The stemmer utilises a single table of rules, each of which may specify the removal or replacement of an ending. This technique of replacement is used to avoid the problem spelling exceptions as described earlier, by replacing endings rather than simply removing them the stemmer manages to

do without a separate stage in the stemming process, i.e. no recoding or partial matching is required. This helps to maintain the efficiency of the algorithm, whilst still being effective [14]

The rules are indexed by the last letter of the ending to allow efficient searching and are of the following form as described in [14]

- An ending of one or more characters, held in reverse order
- An optional intact flag '*'
- A digit specifying the removal total (zero or more)
- An optional append string of one or more characters
- A continuation symbol, '>' or '.'

The following example demonstrate how the rules are stored and used and how replacement can be used to negate the need for recoding. The rule *nois4j>* causes *sion* endings to be replaced by *j*. This acts as a pointer to the *j* section of the rules, leading to the following transformation

provision → *provij* → *provid*

The *j-transformation* is utilised to ensure that the terms *provision* and *provide* are correctly conflated to the stem *provid*. [14]

The algorithm has four main steps detailed below, as presented in [14]

1. Select relevant section;
   Inspect the final letter of the term and, if present, consider the first rule of the relevant section of the rule table.
2. Check applicability of rule;
   If final letters of term do not match rule, or intact settings are violated or acceptability conditions are not satisfied go to stage 4.
3. Apply Rule;
   Remove or reform ending as required and then check termination symbol, and either terminate or return to stage 1.
4. Look for another rule;
   Move to the next rule in table, if the section letter has changed then terminate, else go to stage 2.

The following diagram presents the steps and decisions that are made during the Paice/Husk stemmer.

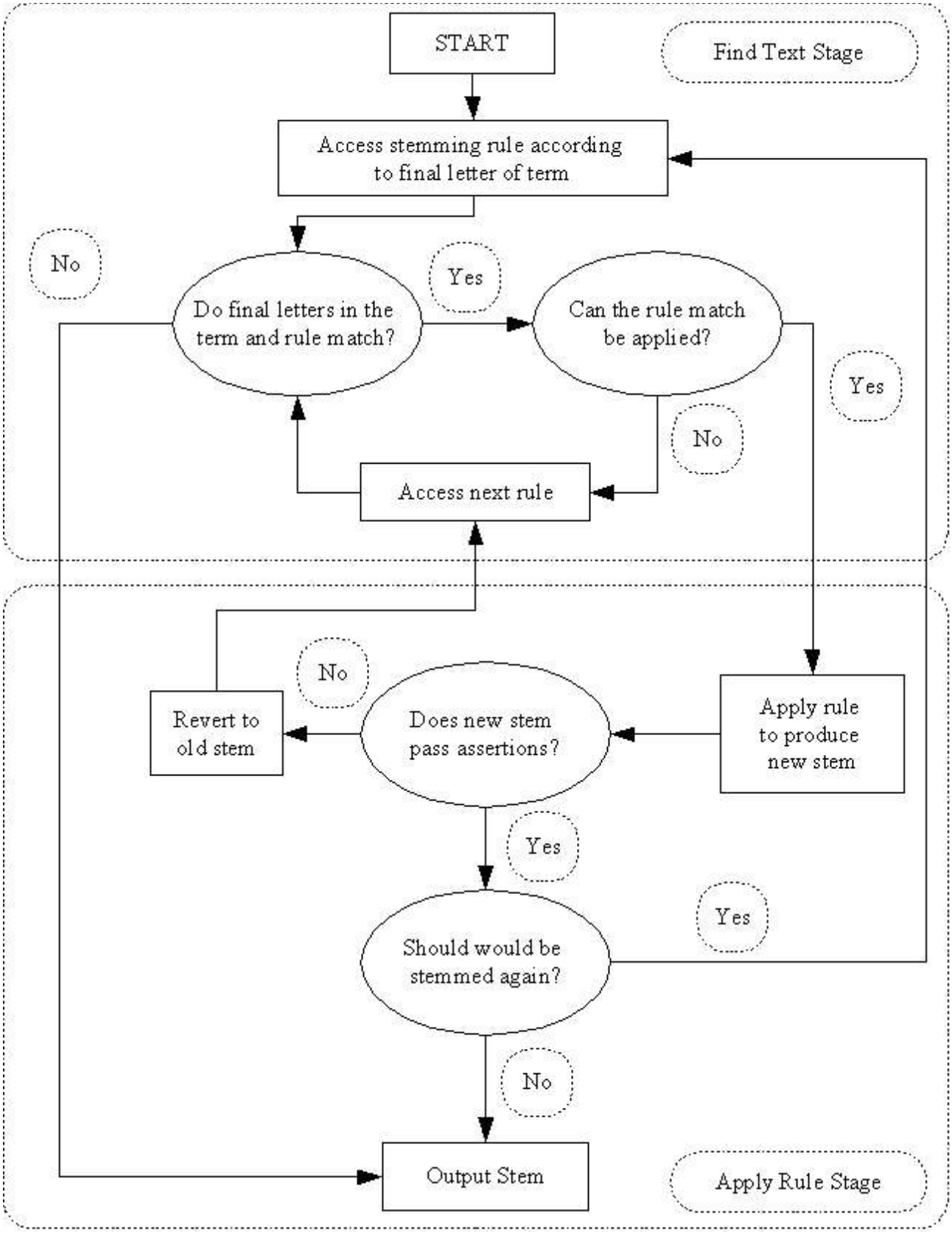

Figure 2.3 Paice/Husk Stemmer [13]

2.2.3) Porter:

The Porter stemmer was first presented in 1980 [19] and was developed by Martin Porter at the University of Cambridge. The stemmer is a context sensitive suffix removal algorithm. It is the most widely used of all the stemmers and implementations in many languages are available.

The stemmer is divided into a number of linear steps, five or six depending upon the definition of a step, that are used to produce the final stem. The following section will describe each step in turn, combined with a diagram displaying all the key steps of the algorithm. However a number of definitions regarding the stemmer need to be made before the steps can be explained. The following definitions are presented in [19]

A *consonant* is a letter other than A, E, I, O, U and Y preceded by a consonant. For example the in the word *boy* the consonants are B and Y, but in *try* they are T and R. A *vowel* is any letter that is not a consonant. A list of consonants greater than or equal to length one will be denoted by a *C* and a similar list of vowels by a *V*. [19]

Any word can therefore be represented by the single form;

$$[C]\ (VC)^{m}\ [V]$$

Where the superscript *m* denotes m repetitions of VC and the square brackets *[]* denote the optional presence of their contents [19] The value *m* is called the *measure* of a word and can take any value greater than or equal to zero, and is used to decide whether a given suffix should be removed. All such rules are of the form;

$$(condition)\ S1 \rightarrow S2$$

Which means that the suffix S1 is replaced by S2 if the remaining letters of S1 will satisfy the *condition* [19].

The first step of the algorithm is designed to deal with past participles and plurals. This step is the most complex and is separated into three parts in the original definition, 1a, 1b and 1c. The first part deals with plurals, for example *sses* → *ss* and removal of *s*. The second part removes *ed* and *ing*, or performs *eed* → *ee* where appropriate. The second part continues only if *ed* or *ing* is removed and transforms the remaining stem to ensure that certain suffices are recognised later. The third part simply transforms a terminal *y* to an *i*, this part is inserted as step 2 in figure 2.4. [19]

The remaining steps are relatively straightforward and contain rules to deal with different order classes of suffices, initially transforming double suffices to a single suffix and then removing suffices providing the relevant conditions are met. [19]

The following diagram illustrates the six steps of the Porter algorithm

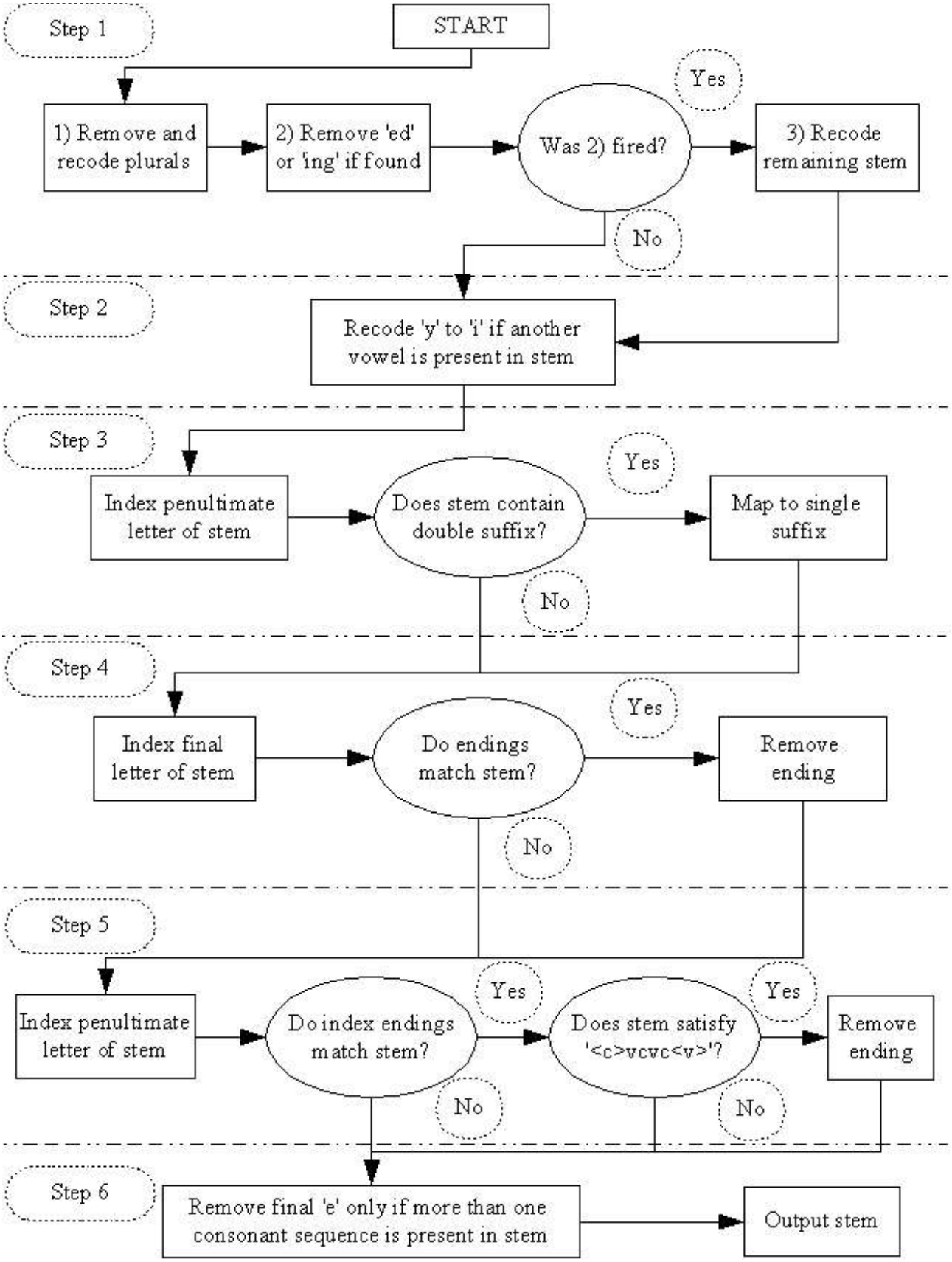

Figure 2.4 Porter Stemmer

2.2.4) Dawson:

The Dawson stemmer was developed by John Dawson and was first presented in 1974 [1]. The stemmer is similar to Lovins as it is a single-pass context-sensitive suffix removal stemmer and was developed at the Literary and Linguistics Centre, Cambridge [1]

The main aim of the stemmer was to take the original algorithm proposed by Lovins and attempt to refine the rule sets and techniques, and to correct any basic errors that exist. The first step was to include all plurals and combinations of the simple suffices, this increased the size of the ending list to approximately five hundred [1]

The second phase was to employ what Dawson called the *completion principle* in which any suffix contained within the ending list is *completed* by including all variants, flexions and combinations in the ending list. This increased the ending list once more to approximately one thousand two hundred terms, although no record of this list is available. [1]

A similarity with the Lovins stemmer is that every ending contained within the list is associated with a number that is used as an index to search an list of exceptions that enforce certain conditions upon the removal of the associated ending. These conditions are similar to the Lovins stemmer in that they may enforce either a minimum length of the remaining stem (with a minimum length of two for all stems) or that the ending can only be removed/shall not be removed when set letters are present in the remaining stem. [1]

The major difference between the Dawson and Lovins stemmers is the technique used to solve the problem of spelling exceptions. As discussed in section 2.2.2 the Lovins stemmer utilises the technique known as *recoding*. This process is seen as part of the main algorithm and performs a number of transformations based on the letters within the stem. In contrast the Dawson stemmer utilises *partial matching* which, as described above, attempts to match stems that are *equal* within certain limits. This process is not seen as part of the stemming algorithm and therefore must be implemented within the information retrieval system. Dawson warns that without this additional processing many errors would be produced by this stemmer [1]

2.2.5) Krovetz:

The Krovetz stemmer was presented in 1993 [9] by Robert Krovetz and is a *linguistic lexical validation* stemmer. It is a very complicated low strength algorithm due to the processes involved in linguistic morphology and its inflectional nature. [9]

The stemmer utilises the process of *dictionary lookup* in order to verify all removals that occur in

the following steps

1. Transformation of plural to singular forms,
2. Conversion from past to present forms,
3. The removal of *ing*,

The dictionary lookup also performs any transformations that are required due to spelling exception and also converts any stem produced into a *real* word, whose meaning can be understood. [9]

Krovetz proposes that due to the high accuracy of the stemmer, but weak strength, it could be useful within information retrieval if used as a form of pre-processing performed before the main stemming algorithm. This would provide partly stemmed input for the stemmer that deals with common situations accurately and effectively, and therefore could reduce common errors [9]

2.3) Other Information Retrieval Techniques:

2.3.1) Prefix Stripping:

All the stemmers detailed in section 2.2 use various techniques to remove suffices from words in an attempt to reduce the word to its morphological root or stem. Many studies, including [4] & [7], have shown that this process can increase the recall performance of information retrieval systems

Prefix removal is another techniques that can be used to reduce a word to its stem, but its effectiveness is less well documented. This question about the usefulness of prefix removal within a stemmer is much more complicated than suffix removal due to the effect a prefix often has on a word.

"Since the removal of a prefix will often radically alter the meaning of a word or else reduce it to a less precise, generically related term" [15]

Within certain special cases errors within suffix removal stemming algorithms can be attributed to the fact that the root morpheme of a word is *trapped* between a prefix and one or more suffices, such as ultranationalism. In these cases the stemmer will often reduce the word to either the prefix or the morpheme with the prefix still attached. This suggests that a prefix removal algorithm used in conjunction with a stemmer could produce more accurate results. [4]

However the usefulness of prefix removal within the English language remains questionable as no inflectional prefixes exist, which could be removed without affecting the words meaning. Derivational prefixes exist, but all can have an affect on the meaning of a word. For these reasons the project has not implemented any form of prefix removal.

2.3.2) Stop List:

A user enters a number of key terms when using an information retrieval system, these will include the main terms that will be searched for, but they may also contain a number of *stop words*. The are common terms that have an impact on the meaning and structure of a sentence but have no inherent meaning themselves. Examples are conjunctions, *and*, and prepositions, *for*. These types of words will often be excluded from the key terms used by the information retrieval system for searching. A *stop list* is simply a collection of stop words that are stored within the system and searched before any word is stemmed.

The benefits of a stop list include improved efficiency, as terms are not stemmed before being excluded, and also unnecessary searches for common words increase the time taken for retrieval. Common words can also have a detrimental effect of the relevance of the results returned as they will be present in many unrelated documents.

In certain circumstances a user may wish to negate a stop list, this can often be achieved by marking a specific term with an inclusion character, such as '+', or by performing a phrase search that considers all terms entered.

2.3.3) Exception List:

An obvious technique that could be used to improve stemmer performance would be an *exception list*. An exception list is simply a collection of known errors that occur when a stemming algorithm is used. The errors are stored along with the correct stem that should be returned and this list can be accessed before stemming occurs and prevent key terms from being wrongly conflated [4]

When an exception list is used within a stemmer it is an inherent risk that it becomes too large and difficult to search to remain an efficient way of correcting mistakes. For this reason it is important that alternative ways of correcting mistakes within the algorithm are investigated before an addition to the exception list is made. These alternatives include

- Adding or removing a suffix from the ending list,
- Adding or modifying an exception rule,
- Modifying the list of recodes or the limits used for partial matching

This technique has the advantage of fixing specific errors within the system and ensuring perfect accuracy whilst doing so, it also can be constantly updated by the users/administrators of a system as more errors are uncovered during daily use. This will allow the information retrieval system to *evolve* and become more accurate the more it is used.

3) Previous Work:

Previous work investigating the performance of the Lovins stemming algorithm has been reported [5] and the following section will describe the main work undertaken and results found. The work focussed on implementing the Lovins stemming algorithm presented in [11] and then evaluating it uses the techniques presented in [16]

3.1) Implementation of Lovins Stemmer:

The implementation of the Lovins stemming algorithm as described in the original proposal was implemented in Java, contained within the class *LovinsStemmer*, and utilises three other classes,

- *Ending*, simple structure class used to store the endings and their associated rules.
- *Recode*, used to store a recode, including the old and new endings and any conditions placed on the transformation.
- *Output*, the class is used to store information on the steps taken when a word is stemmed and then to print this information to a file.

Two text files containing the endings and recodes are also used, the information is read in during the initialisation stage.

The initial aim was to store all the endings, recodes and exceptions in text files and read them into the program, this was to prevent recompiling when a change was made to the rules of the stemmer. However this was not possible with the exceptions because no generic form covers all exceptions, thus making it extremely difficult to read the rules in from a file. For this reason they were coded into the *LovinsStemmer* class.

The class was designed to perform each individual step within the algorithm, as illustrated in figure 2.2, to allow for rigorous testing of the individual components. By isolating the steps within different methods changes/optimisations can easily be made to components within the class without affecting the other steps.

3.2) Evaluation Techniques:

The project utilised the method for stemmer evaluation proposed by Paice [16], which calculates performance metrics based upon the number of individual errors that occur when the algorithm is implemented. There are two types of error which occur within all stemming algorithms, these are *understemming errors* and *overstemming errors*. The first occurs when two words that ought to be merged together are not, the second is when two words that should remain distinct are merged. [16]

Due to the irregularities in natural language it is not plausible to expect perfect accuracy from a

stemming algorithm. The aim when developing a stemmer is therefore to take all possible steps to limit the number of errors that occur. For this reason the evaluation method attempts to differentiate between different stemming algorithm performance by counting individual errors and computing results based on these figures. [16]

The motivation for the development of stemmers was to improve information retrieval performance by conflating morphologically related terms to a single stem. This is formally defined as

"A stemmer should conflate all and only those pairs of words which are semantically equivalent and share the same stem" [16]

This definition creates the problem of how the program will decide when two words are semantically equivalent. The solution was to provide input to the program in the form of *grouped* files. These files contain list of words, alphabetically sorted and any terms that are considered by the evaluator to be semantically equivalent are formed into *concept* groups. Whether two words are related in this way can often be debatable and could lead to discrepancies based upon the opinion of the user. To assist the user words can be grouped together either *strongly* or *weakly*. This is accomplished by inserting *weak barriers* between terms that the evaluator considers loosely connected. It is felt that a this, combined with a suitably large word set, c10,000 words, will negate the effects of any subjectivity when calculating the results. [16]

The following section will explain the various metrics that are produced by the evaluation method, and the calculations required to obtain them;

- Global Wrongly-Merged Total, GWMT:
  After the input file has been stemmed in it necessary to assign a number to each word, which corresponds to the concept group the original term belonged to. Once these numbers have been assigned the words will be sorted into a set of *stem groups*, which contain all terms that have been conflated to the same stem. If all the terms within a stem group belong to the same concept group then no over stemming errors have occurred within that group. However if the stems are derived from *f* different concept groups then the *wrongly-merged total* for the group can be defined as;

  $$WMT_s = 0.5 \sum_{i=1}^{f} n_{si} (N_s - n_{si})$$

  Where N_s is the total number of stems in the group and n_{si} is the number of stems derived from the i^{th} concept group. Summing the WMT values across all stem groups gives the *global wrongly-merged total*. [16]

- Global Actual Merge Total, GAMT:
  The above value is normalised according to the number of actuals word pairs which become identical during the stemming process by defining, for each stem group, the *actual merge*

total;

$$AMT_s = 0.5\ N_s\ (N_s - 1)$$

Where N_s is as above. Summation over all groups gives the *global actual merge total*. [16]

- Overstemming Index, OI:
  The *overstemming index* is simply defined as the ratio;

$$OI = GWMT/GAMT$$

  The value ranges from zero to one, with a lower value representing better performance. [16]

- Global Desired Merge Total, GDMT:
  After the initial stemming of the file the words are still contained within the concept groups defined by the user. For each concept group *g* the *desired merge total* is defined as follows;

$$DMT_g = 0.5\ N_g\ (N_g - 1)$$

  Where N_g is the number of words in the group, summing over all groups the *global desired merge total* is obtained. [16]

- Global Unachieved Merge Total, GUMT:
  If within a concept group all terms have been conflated to the same stem then no understemming errors exist. However if one or more pairs of words have not been merged correctly then the *unachieved merge total* for the group *g* is defined as;

$$UMT = 0.5 \sum_{i=1}^{f} n_{gi}\ (N_s - n_{gi})$$

  Where *f* is the number of distinct stems in the group and n_{gi} is the total number of cases of stem *i* in the group. Summing over all groups give the *global unachieved merge total*. [16]

- Understemming Index, UI:
  The *understemming index* is defined as the following ratio;

$$UI = GUMT/GDMT$$

  As with the OI the values range from zero to one, with lower values indicating less errors [16]

- Error Rate Relevant to Truncation, ERRT:
  The understemming and overstemming indices are measurements of specific errors that occur during the implementation of a stemming algorithm They provide a good indication to the accuracy of the stemmer, however they cannot be considered individually during results analysis. The values of the indices are interlinked as a light stemmer will produce very few overstemming errors, whilst many understemming errors will occur. Therefore either value taken separately from the other could provide an unbalanced view of a stemmers performance. [16]

  For this reason it was necessary to produce a composite measure of the two values that could

be judged against a given baseline. The baseline was obtained by performing the process of *truncating*, reducing a word to a given fixed length, the input file and then calculating the understemming and overstemming indices for a range of truncation values. These values are then plotted onto a graph, as shown in figure 3.1, to produce a *truncation line*. [16]

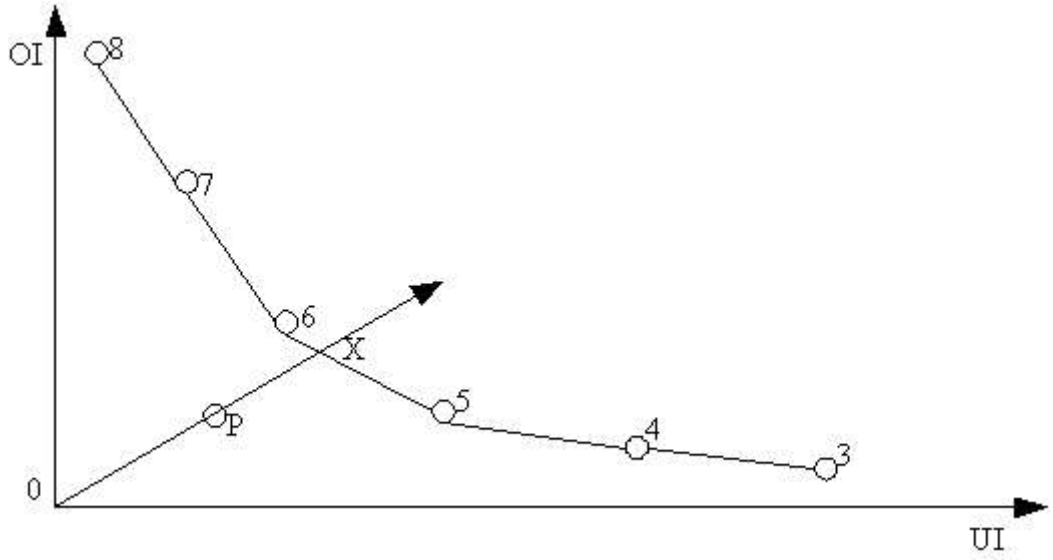

Figure 3.1 ERRT Graph [16]

Given the point P for a non-truncation stemmer, by extending the line OP until it intersects the truncation line at the point X the *error rate relevant to truncation* can be defined as

ERRT = (length OP)/(length OX)

This value is used as the main value for comparison of stemmer performance [16]

3.3) Development of AWT GUI:

The original GUI that was developed provided the following functionality, as presented in [5];

- Selection of a file to be grouped
- Preparation of input file for grouping
- Specifying the name of the grouped file outputted
- Importing of a pre-grouped file
- Selection of the stemmer to be used
- Performing the stemming of the grouped file
- Calculation of the metrics mentioned in section 3.2
- Logging results to a chosen file

The aim of the GUI was to provide the evaluator with a user-friendly interface that collated the functionality of all the classes and packages developed, and as much as possible automate the process of gathering results. The interface controls the instantiation of all other classes used in the evaluation process and passing the information between them, this removes the responsibility from

the user. The only step within the evaluation procedure that still requires significant user input is the creation of a grouped word set; this process will be discussed later in section 6.

3.4) Packages Developed:

This section will describe the packages, and classes within them, that were developed during the previous project. The following descriptions are adapted from [5]

- Utilities Package;
  The *Utilities* package was developed to provide the functionality required during the grouping procedure and certain evaluation steps. All the classes deal with text file input and perform transformations on the information before outputting to a new file.
  - *Cleaner;* this class is designed to standardise an input file by removing all white space, converting all word to lower-case and placing each term on an separate line.
  - *FileSort;* this class takes a cleaned word file as its input and sorts the terms alphabetically using an insertion sort.
  - *Deduplicator;* performs the removal of duplicate terms in the input file by taking the sorted words and comparing adjacent terms and removing one if they match.
  - *GroupWords;* accomplishes all task required during the grouping procedure, including the reading/writing of terms to and from the input/output files, insertion of barriers when required and providing a suggested action to the user .
  - *Regroup;* this class assigns numbers to all terms according to the principle of concept groups as discussed above. Two output files are required, one where weak barriers are ignored and one for when they are treated as strong barriers.
  - *StemmedFileSort;* using an insertion sort this class reforms the word set into the stem groups described earlier.
- Evaluation Package;
  The *Evaluation* package contains classes to calculate all the indices described in section 3.2, and provide any extra functionality required for these calculation.
  - *Comparison;* performs the truncation of the input file to allow for the ERRT value to be calculated.
  - *OverCalc;* takes four files as input (as the barriers have been removed) for numbered and regrouped output from the stemmer. This allows efficient calculation of overstemming indices relating to both concept and stem groups.
  - *UnderCalc;* requires only a single input file, taken as the direct output from the stemmer, as the barriers remain and the class can calculate the understemming related indices for both ignoring weak barriers and treating them as strong.
- Stemmer Package;
  The *Stemmer* package contains all classes that are required to implement the Lovins stemming algorithm together with the user interface. This is discussed in greater detail in section 3.2

- *Ending;* stores an ending and its associated rule.
- *Recode;* stores the ending before and after a recode and any conditions.
- *Output;* contains information about the steps taken by the stemmer in producing the output, also implements a *toString* method for easy output of this data.
- *LovinsStemmer;* main class containing the implementation of the original algorithm proposed by Lovins.
- *UserInterface;* implemented using the standard AWT to provide the functionality discussed above

3.5) Results and Conclusions:

The overall aim of the earlier project was to investigate the effects of making the modification of introducing an iterative step into the Lovins algorithm in an attempt to *complete* the ending list, as described by Dawson [1]. The motivation was to decide whether the simple change of running the algorithm twice, or multiple times, could compensate for the known incompleteness of the ending list. The step was implemented and test run on a grouped file, c10,000 words in size, producing the following results.

	Weak Barriers Ignored		Weak Barriers = Strong	
Stemmer	ERRT	SW*	ERRT	SW*
Lovins	91.10	0.0341	73.29	0.3012
Lovins 2-pass	86.08	0.0841	69.28	0.5246
Lovins n-pass	83.02	0.1033	70.05	0.6611

Table 3.1, Previous Results [5]

*SW (Stemmer Weight) is defined as OI/UI

The results showed that slight improvements in both a 2-pass and n-pass implementation of the Lovins algorithm were achieved. The results were not fully verified, due to time constraints, and were presented as provisional due to discrepancies between them and the original results presented in. [16]

Although provisional the results were promising as they showed that improvements in stemmer accuracy could be obtained with simple modifications to the algorithm. The gave the motivation for this project to verify the results and take a more systematic and refined approach to improving the Lovins algorithm to see if better results could be obtained.

4) Verification of Correctness:

4.1) Previous Verification:

As mentioned in section 3.5 the results of the original program were presented in [5] as provisional due to discrepancies with the original results, despite the same grouped word file being used. The original project performed testing on both the correctness of the stemming algorithm and the calculations for the performance metrics.

When performing the verification procedure for the stemming algorithm it was deemed unnecessary to test every possible combination of endings, exceptions and recodes, as the methods remain the same for each removal, condition and recode. Therefore a representative selection of ending removals, with associated condition rules, and final recodes was chosen and the stemmer was run on this reduced word set, c.200 words, and the correctness was confirmed via manual inspection [5]

Demonstrating the correctness of the calculation procedure was a more complicated procedure. Originally the project planned to verify the correctness based on the results presented in [16], as the same input file was available for use. However when it was not possible to obtain the same results a second method for testing was required. This involved calculating by hand the expected results for a small section of the input file, twenty words in length. The program was then run and the results matched the hand calculations, but due to constraints on time it was not possible to perform further tests and the calculation were assumed to be performing correctly. [5]

4.2) Current Verification:

Obviously one of the first aims of the present project was to perform extended testing on both the original implementation of the Lovins stemmer and the evaluation calculations. A secondary aim of the verification of the original programming was to provide a collection of test data and expected results that could be used to verify any future modifications to the program, and also sets of data that could be used to ensure the correct operation of the Paice and Porter stemming algorithms that are to be implemented and integrated into the evaluation system.

The first step of the extended verification process was to ensure the correct operation of the Lovins stemming algorithm. This involved expanding the representative selection discussed in 4.1 to include more combinations of endings, exceptions and recodes. This further testing indicated that the implementation was working in accordance with the original algorithm specifications

The Paice and Porter stemmers are implemented in the updated system and the correctness of the implementations would also need to be demonstrated. The exact implementation techniques will be

discussed later in section 5.5 but both were verified as working correctly. The Paice implementation was taken from [13] and manual inspection of a reduced word set ascertained that the program was operating to the specifications of the original algorithm. The Java implementation of the Porter algorithm was obtained from [W2] and was supplied with two word lists, the second of which contained the expected results when running the Porter stemmer on the first. This allowed for a simple demonstration of the implementation's correctness.

Verifying the correctness of the calculations proved difficult and time consuming. Three word sets, two c.100 and one c.500 words, were obtained as subsets of the original word set. For each of these files the correct output for each phase of the calculation procedure was created, including the sorting, numbering, regrouping and all indices calculated by the system. Once the expected values had been calculated each step taken by the system was individually validated.

Systematically ensuring the correct output is given at each stage of the evaluation process is necessary as it is too difficult to discover the source of errors in a system where the results outputted from a given phase are entirely dependent on the input received from the previous step. This dependency on previous manipulation of the data led to latent errors remaining in the system and not becoming apparent until the results from calculations performed later in the program were displayed.

One such error that was discovered involved the sorting of the stemmed and numbered file into stem groups, as discussed in section 3.2. The word set was sorted correctly into stem groups, but the calculation program assumed that the elements within these stem groups were sorted by concept group, which led to later errors in the overstemming calculations. This error was addressed, by altering the sorting of the word set into stem groups to also sort them by concept group, rather than alter the calculation programs.

The second error was identified within the OverCalc class and was causing errors within the WMT calculation due to a loss of precision. The error was caused by not casting the result of dividing the summation by two as a *double* and hence the accuracy of the calculation was lost. This fault was rectified by inserting the required code to cast the result and ensure the calculation's correctness.

Once these faults had been fixed the program was run on the full word set and the results matched those that were presented in [16]. This result set could now be used to test the program whenever any modifications to the code were made.

5) Optimisation and Additions to the Current Tool:

5.1) Updating the Interface

In the original system the the GUI was developed to provide a user friendly front end that allowed the user access to the functionality of all the classes within the system from one screen. The original GUI was developed utilising the AWT classes of the Java API. The following is a screen shot of the original GUI in its original form.

Figure 5.1 AWT GUI [5]

This GUI performed effectively but after persistent use of the system certain enhancements were desired. These included the following

- Ability to choose the file to be grouped/imported from a list to limit the possibility of *FileNotFoundExceptions* arising from typing errors.
- Increase size of *Current Word Group* and *Results* text fields to allow more data to be displayed.
- Unify the layout, by ensuring buttons, text fields, text boxes, combo-boxes and labels adhere

to set dimensions and align consistently.

- Increase the overall size of the GUI to allow for more spacing between buttons to prevent errors occurring during use.
- Improve the *look and feel* of the program by including graphics and enhancing the colour scheme.

These improvements required the re-writing of the interface code to implement the Java Swing API, and resulted in the following GUI being developed.

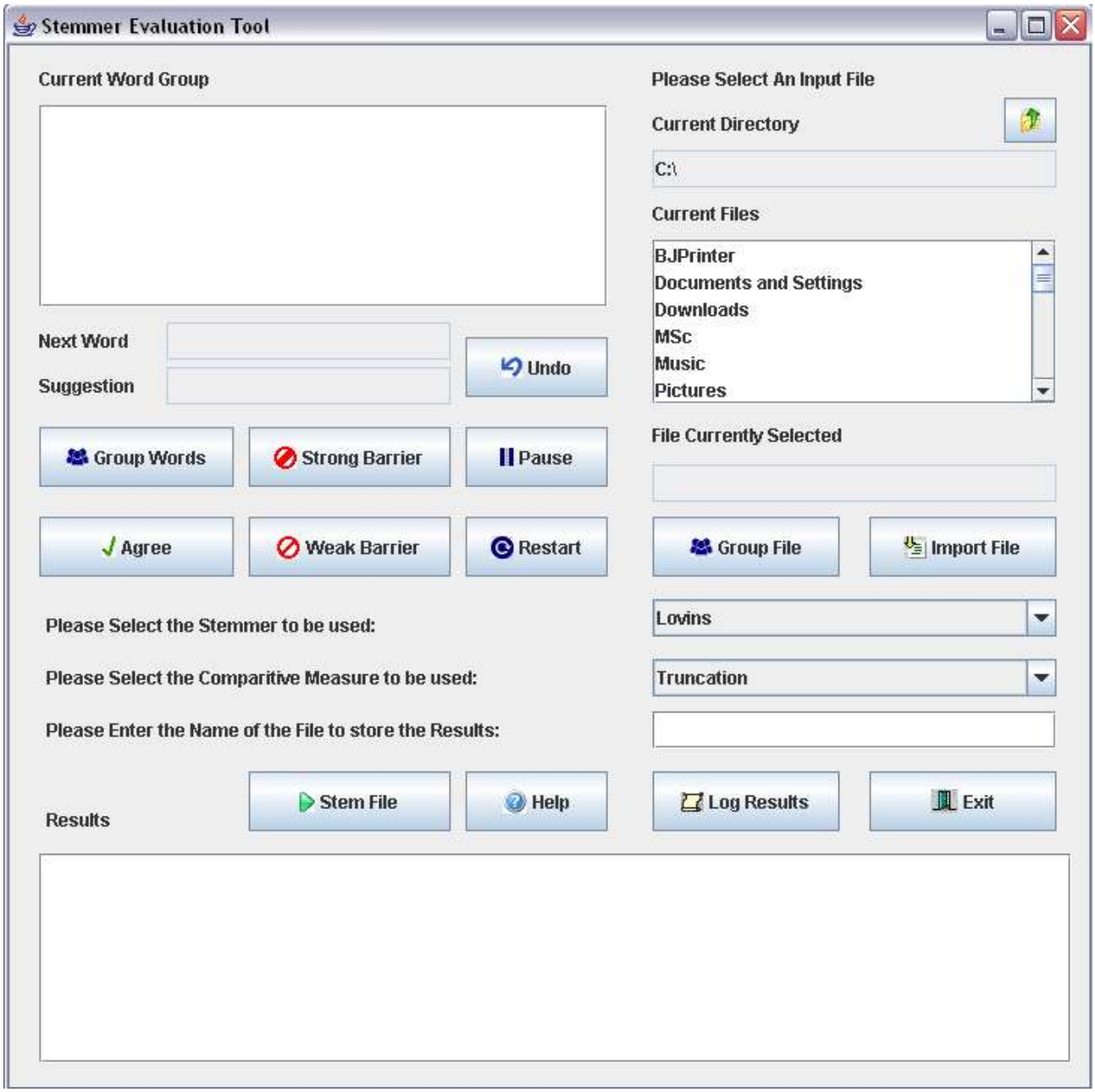

Figure 5.2 Swing GUI

5.2) Improving Efficiency:

Following the procedures described in section 4 the correctness of the system was assured, but a problem of efficiency still remained. The sorting algorithms within the system are all insertion sorts, as the extra memory required for the algorithm is readily available and it is relatively simple to implement. The algorithm is known to have a time complexity of $O(n)$ in the best case, when the list is already sorted, and $O(n^2)$ in the average and worst case. [W1] This efficiency was considered acceptable and when the system was run the first sorting performed on the input file to be grouped was completed quickly. However subsequent sorting stages would take a lot more time to complete. The following is pseudo-code of the sorting algorithm taken from the program used to regroup the file into stem groups

```
Initialise two lists one containing the input the second empty
Place first element of input into output list
while still more input
       Obtain next element from input
       while element has not been placed into output list
              if element to be input is < next element in output insert into output
              else obtain next element in output for comparison
              if end of output reached insert element
       Check for end of input
Return output list
```

This code will sort the input vector one element at a time into the output vector. The problem arises when the input list is already sorted or is very close to sorted, as occurs when regrouping into stem groups after stemming. In this situation the program will have to scan the whole word list when sorting each individual term, as the word will most likely need to be inserted at the end of the list. This problem was solved by rewriting the implementation of the insertion sort to begin at the end of the word set. In the case described above this will reduce the number of comparisons required to sort a 10,000 word list by approximately 5×10^7. This improvement is reflected by the greatly reduced time taken to perform the calculation of the indices.

The efficiency was also improved in the Utilities package by removing the *FileSort, Deduplicator, Cleaner* and *GroupWords* classes and incorporating the functionality into the new Swing GUI. This removed the need for the instantiation of new classes and the reading and writing of information to and from files. The efficiency was also improved by incorporating the removal of multiple matching terms from the input into the cleaning stage, removing one loop through the input from the program.

5.3) General Debugging:

Faults in the code that led to errors in the displayed results were addressed as discussed in section 4, but two bugs in the code remained. The first involved the retention of temporary files following the closure of the program and the second focussed on the progress bar display during the stemming/calculation phase of the evaluation.

The non-deletion of temporary files has no detrimental effect upon the system's performance but causes inconvenience to the user and so the code for all classes that created temporary files for the passing of information during the calculation of the performance metrics was re-written to mark all these files to be deleted upon exiting the program. This highlighted one problem with the class calculating the overstemming indices not closing the buffer used to read from the files, causing the files to not be deleted. This was a simple change to the code and all files are deleted as desired

The second bug involving the progress bar was a more complicated problem and remains unresolved. The implementation of a progress involves the creation of a new *JFrame* which holds the progress bar within it and appears on screen when the stemming and calculations are being performed. The frame appears as expected but the progress bar is never drawn onto the screen. After much investigation the problem appears to be related to the threading of the program.

As the GUI is designed to provide all functionality from one screen it was seen as unnecessary to create the screen in a different thread to the main program. This means that when the program is performing computationally intensive tasks the *repainting* of the screen is delayed, which means that although the frame is made visible the progress bar is not *drawn* on the screen. To re-write the code into a multi-threaded application would have been too time consuming and could have introduced new errors into the system.

For this reason the frame was instantiated and the progress through the different stages of the program was communicated to the user via the changing of the frame's title. This replaces the main functionality of the progress bar, assuring the user the system is still running, as the title changes consistently due to the large number of different stemming/calculation stages in the system.

5.4) Extra Functionality:

The following section will describe the extra functions, excluding the implementation of additional stemming algorithms, that were added to the original system.

- *Pause/Restart;* Allowing the user to pause, and then later restart, the grouping procedure was identified as a requirement during the development of the original program, and provision for

it was made in the AWT GUI, see figure 5.1. This functionality was considered necessary as the grouping process is time consuming and can take many days. Time constraints during the original system development meant that it was not possible to implement this functionality. For this reason when the GUI was re-written the ability to pause and restart was added to the system.

When the user begins the grouping procedure on a file the information is read into a vector and *cleaned*. The user will then perform a number of *grouping actions* on the words, e.g. inserting barriers. The result will be stored in another vector, ready to be written to the output file. When the user wishes to pause the grouping the string 'PAUSE' is added to the output vector and this is then outputted to the file, followed by the ungrouped portion of the input vector. By placing all the information regarding the current state of the grouping procedure to a file it allows the user to group a word list over multiple system sessions.

When restarting, the file is accessed and everything before the 'PAUSE' string is added to the output vector and everything following to the input vector. This restores the internal state of the program to the point before the procedure was paused, all that remains is to update the external state, the GUI visible to the user, to reflect the changes.

- *Undo;* A main consideration during the development of the GUI was how to allow the user to easily reverse their actions in the event of a mistake. An undo button was required and needed to allow the user to revert to the system state as it was before the last button press.

  The undo function is implemented in the Swing GUI class by maintaining a variable representing the identity of the last button pressed. The program can then use this value to identify the last grouping action performed and undo the action as required. This GUI can then be updated to to show that the state has been reverted. This reduces the need for the user to manually alter any mistakes in the output file.

- *Help;* To assist new users a help screen was added into the system. This is called by pressing the on-screen button which will display a new frame containing a text field with explanations on how to perform all possible functions within the program.

- *ERRT;* The original system calculated the overstemming and understemming indices for the chosen stemming algorithm together with a *comparative measure*, this was the performance metrics for a chosen truncation value. To calculate the ERRT value discussed in section 3.2 the user would have to manually create the graph to retrieve the value. The new system includes a class called *ERRT* that takes as its input parameters a series of four arrays containing the overstemming and understemming indices for the chosen stemmer and

truncation values ranging from three to eight, when weak barriers are ignored and when they are treated as strong.

The program then uses a series of simple mathematical formulae to calculate the gradients, intersection points and lengths of the vectors within the ERRT graph. This allows the ERRT value to be determined by the system and displayed along with the other results.

5.5) Addition of Stemming Algorithms:

One of the primary aims of the system detailed in section 1.2 was to add additional stemming algorithms to the evaluation program. The motivation for this extension to the system is to enable the user to perform tests on all common stemming algorithms allowing for comparative results to be obtained.

The updated system allows the user to select which stemming algorithms is to be tested from within the GUI. The program requires an interface called *Stemmer* to which all algorithms conform, this allows the system to remain unaware which algorithm is being used and utilise a set of standard method calls to act upon the stemmer.

Java implementations of the three main stemming algorithm, Paice, Porter and Lovins were all available from external sources. The Paice stemmer was taken from [10], the Porter stemmer from [W2] and the Lovins stemmer from the original project [5]. These implementations were modified to conform to the defined interface and the GUI was modified to instantiate the chosen algorithm. The correctness of the implementations, including the modifications, was verified utilising the data sets made obtained as described in section 4.

To enable the proposed optimisation of the stemmer's structures were needed that supported the *retention of information* described in section 1.2. This process involves storing data about the steps taken by the stemming algorithm in reducing the original term to its stem, including the original and new terms, endings removed, exceptions fired and recodes performed. This information is stored in a number of structure classes, one for each stemmer, which facilitate the storing and outputting of the information. The techniques that are used to garner useful information from this data are detailed later in section 7.

6) Production of Grouped Word File:

6.1) Motivation:

The method for evaluating stemming algorithms presented in [16] relies upon the creation of *grouped word files*. This *grouping* procedure involves collating all morphological related terms and separating these *concept groups* with *barriers*. These barriers may be either *weak* or *strong* depending on the preference of the evaluator.

A number of grouped word files were available during the previous research, however the largest of these was c.10,000 words and all of them were created by the same person. These facts have motivated the creation of a larger grouped word set to add a greater a level of validity to any results gathered. This new word set was compiled from a number of word lists used in Scrabble® word checkers. These were chosen as they list a large number of word variants and focus on common terms used in the game. The list is c.20,000 words long and was grouped using the Swing GUI over a period of two weeks.

6.2) Problems and Challenges:

The process of creating a grouped word file is a complex and time-consuming operation that is inherently error-prone due to a number of factors that will be discussed in this section. These problems need to be dealt with carefully and in a standardised manner, which will assist in the reduction of any incorrect relations within the word set. The following lists all the issues that arose during the creation of the new grouped word file;

- Time Consuming: The process of manually grouping a list of c20,000 words took almost two weeks and is very repetitive. Concentration was an issue that needed to be addressed and ample time needed to be set aside to perform the grouping procedure to allow for the taking of regular breaks.
- Disjoint Concept Groups: A problem that arose during grouping, and that is also discussed by Sanchez [21], is that words may only belong to a single word group. Often words will be related to two or more other terms that are themselves unrelated. The generalised method presented in [21] uses a thesaurus to place terms in as many concept groups as required. Although this method would remove the need for manual grouping of files, which the author considers prohibitive to gathering large sets of evaluation data, it requires the modification of the evaluation calculations. The problem is countered by using the concept of weak and strong barriers to enable the system to differentiate between closely and loosely related terms.
- Polysemes: Any word list will contain a number of Polysemes, defined as words which have multiple related meanings. A problem for the user grouping the file is deciding how related

the derivatives of these multiple meanings are and whether they belong to the same concept group. An example of this is the word *right* which has sixteen meanings define in [18] (six adjective, three adverb, four noun, two verb and one exclamation). Eight derivatives are explicitly listed together with the standards inflexions. Thirteen other terms beginning with the stem *right* are also listed together with their derivatives. In this case it is impossible to ensure all terms within a grouped are related for all meanings of the original word. In these circumstances it was decided to group all the terms together as it is not possible for the stemmers implemented to consider the semantic context in which a word is used.

- American Spelling Variations: The word list contains both British and American spelling variations of words. In [11] Lovins explicitly mentions American spelling variations of English words as one of the major causes of spelling exceptions. The recode procedure was therefore designed to correct any errors caused by these variations. As the stemmers are designed to handle them, American and British variations have been placed within the same concept group throughout the grouping procedure.
- Cardinal Numbers: It could be argued that in certain circumstances ordinal and cardinal numbers, such as *six* and *sixth* share the same stem and are semantically related, and therefore should be placed within the same concept group. However, although all pairs of cardinal and ordinal number are definitely related they do not always share the same stem, e.g. *one* and *first*. To maintain continuity within the grouping procedure it was decided that no pairs would be collated together.
- Related Terms Separated: A recurring problem during the grouping procedure is that semantically related words may be separated by unrelated terms, e.g. *age* and *ages* will be separated by *agent*. The original word file is sorted alphabetically in an attempt to place related terms together but due to the large number of words in the file, many terms that are related are kept apart. This means that they cannot be grouped together from within the GUI and the user needs to keep a record of these mistakes and correct the file manually. A better solution would be to change the *current word group* display within the GUI to display all words in the file that have been grouped and to enable dragging and dropping of terms within it.

Overall these problems were dealt with as explained but the inherent problem of subjectivity is less easily overcome. When grouping a file it is not always clear whether two terms are semantically related or not, and if this relation is weak or strong. The decision will be based on the opinion of the evaluator creating the file, and may differ from person to person. When deciding if two words are weakly or strongly related a standard way of deciding on which grouping action should be performed was required. This method utilised the Concise Oxford Dictionary[18] and grouped a word together strongly if it is explicitly listed as a direct derivative, or is a standard inflexion, and terms were weakly grouped otherwise. This, combined with using multiple large word sets manually grouped by different evaluators, should negate the subjective nature of the procedure and provide valid and insightful results.

7) Optimisation and Refinement:

7.1) Optimisation Package

The aim of this project was to investigate the possible optimisation of the Lovins stemming by utilising information gathered about the steps taken in reducing a word to its stem in the Lovins, Paice and Porter algorithms. To achieve this the system needed to firstly implement the additional algorithms and related structures to output to a file all the information about the stemmers' steps in conflating a word.

Once this data has been collected it needs to be processed to provide information that can be used to improve the rule sets of the Lovins stemmer. This processing is performed by a new package incorporated into the system, called *Optimisation*. The package contains five classes which are listed and explained below;

- *CollateEndings;* This class was used to process the raw data produced during the stemming procedure. Input is provided as a single text file containing the output produced when the Paice or Porter stemmers are run on the grouped file. The program outputs two files, one containing all apparent *unconditional* ending removals and one containing *single conditional* removals

  Within the stemmer output an ending which is removed without the need for context rules to be satisfied is marked as unconditional by preceding the ending with the string *rem:*. These endings are gathered by examining the step(s) taken when producing a stem, and if the step(s) are either a single, or a sequence of, unconditional removals then the total ending removed is added to the collection, and associated with the default Lovins exception rule.

  The single conditional endings involve all stems that are obtained in one step by the stemmer from the original word, and these steps are not preceded by the remove flag in the output. Within the output these endings appear as *transformations* with the original ending and new ending displayed either side of a separating character, e.g. *sses>ss* from step one of the Porter algorithm.

  Only these types of endings are gathered as they are of a format that allows for easy insertion into the Lovins stemmer rule sets, as the single pass model of the algorithm prohibits the addition of endings that are removed in multiple conditional steps

- *SortEndings;* The output from the CollateEndings class produces a number of files containing endings that need to be added to the original rule set. This class provides the functionality to

merge two or more lists of endings into a single list sorted first by length and then alphabetically. This program is used to insert only the unconditional endings, as they have been formatted to be associated with the default exception and therefore require no manual input.

- *RankEndings;* This class is used to rank all endings used within the Lovins stemmer by a proportional error rate, derived from the number of times the ending is removed and the number of errors it is *involved* in. The definition of the error rate raises the question of when is an ending involved in an error. An ending may be involved in one of two types of error, understemming and overstemming errors.

  To identify understemming errors the *correct stem* within a concept group must be identified. The program sets the correct stem within a group to that which appears most often, but if two or more stems are equally common within a group then a result of undecided is returned. For overstemming errors the *correct number*, assigned when calculating the GWMT as discussed in section 3.2, within a stem group must be obtained, again this is set as the most common, or undecided.

  The program has been designed to remain conservative when identifying errors. This is why a result of undecided is returned when it cannot be certain of the correct stem/number. To further increase the accuracy, the errors that are identified as understemming errors are only searched for when weak barriers are set as strong, and vice versa overstemming errors are only counted when weak barriers are ignored. When weak barriers are ignored, the program will consider more words to be related and therefore more understemming errors will occur, and conversely for overstemming errors. By counting the errors as above these 'extra' errors are ignored and only those that are present both when weak barriers are ignored and when they are treated as strong are registered in the ranking process.

- *EndingRank;* The class is used by the RankEndings program to store an ending removed and the information gathered about it. This includes times used, which is recorded as twice the number of times the ending is removed as each time it is used it may be involved in one understemming and one overstemming error, the number of understemming/overstemming errors an endings is involved in and a count of the instances when the program was undecided. This information is used to calculate the *error rate* defined as the total number of errors divided by the number of times used. To avoid giving undue weight to endings which occur very rarely five is added to the number of times used when performing the calculation.

- *EndingUsage;* This program is used to output all the instances of an ending being removed to a file specified by the user. The information is outputted in the same format as the stemmer

output, with the original line number appended to the beginning of the line. This class is used to help the evaluator understand the consequences of any changes to the rule sets by seeing all groups of words that will be affected

All these classes are implemented with simple command line interfaces and are called individually by the user to perform their specific task.

7.2) Refinement Steps Taken:

The following section will detail the steps taken in the attempt to improve the Lovins rule sets. Each step will contain a list of any modifications to the original rules and will be presented alongside the interim results that were obtained. Within the results tables, the headings *Weak Ignored* and *Weak = Strong* will represent the cases when weak barriers are ignored and when they are treated as strong respectively. *Word List A* refers to the original grouped word file containing c.10,000 words taken from a number of document abstracts from within the Information Science domain. *Word List B* refers to the newly created grouped file of c.20,000 words compiled from words lists used in Scrabble®. Each optimisation step is explained and presented together with any interim results

7.2.1) Step 1:

The first step was to use the unconditional endings gathered from the Paice and Porter stemmer output by the CollateEndings class. These were collected from the word lists A and B and the four output files were merged together with the original ending list. During the merge any duplicate endings were removed with priority given to those from the original list, to preserve any associated exception rules.

This process increased the total number of endings within the Lovins rule set from 294 to 1189. The results of this change are presented in table 7.1. As expected although a number of the errors identified within the Lovins stemmer were addressed a large number of new errors were introduced. This initial step was designed to introduce all possible endings into the rule set, the removal of erroneous entries will be performed later to improve performance.

	Weak Ignored	Weak = Strong
Word List A	108.80	111.82
Word List B	113.10	116.35

Table 7.1 Optimisation Step One Interim ERRT Results

7.2.2) Step 2:

The second step was to process the single conditional endings that were output from the Paice and Porter stemmers. These could not simply be added into the ending list as the associated exception needed to be considered. Fifty endings of this nature collected and each needed to be individually evaluated before being added to the system. Many of them were already present within the original ending list, the remaining conditional endings were best modelled within the Lovins stemmer as recodes. These were all implemented and the following four gave consistent improvements across all ERRT values and were added to the recode list.

ript → rib *sses → ss* *bil → bl* *bly → bl*

One further action was taken in this step which involved how to deal with a terminal *i* or *y* that is present in a stem. Currently the Lovins stemmer performs no recoding of either terminal, the Paice stemmer includes the rule *i → y* and the Porter stemmer the rule *y → i*. Each method was tested and the results were consistently better when the *i → y* rule was included, thus it was chosen for the new rule sets. These changes gave the following results;

	Weak Ignored	Weak = Strong
Word List A	106.56	109.32
Word List B	111.19	114.49

Table 7.2 Optimisation Step Two Interim ERRT Results

7.2.3) Step 3:

Having added a large number of endings and increased the number of recodes within the rule sets the next step focussed on the removal of poorly performing endings. The technique for discovering which endings were candidates for either removal from the rule set or modification utilised two classes, RankEndings and EndingUsage. The first program examines the output from the Lovins stemmer and produces a file listing all the endings used sorted in ascending order of error rate. The second program then allows endings that are poorly ranked to be investigated further by viewing all instances in which they are used.

This first round of ending examination focussed on endings that were not only ranked poorly, but were also commonly removed. This lead to the following endings being deleted;

bed ded fed ged med ned ped red ted ved bing ding fing ging ming ning ping ring ting ving anent ared erial istan ler lant lying ogen

These removals led to a major improvement in the results as shown below.

	Weak Ignored	Weak = Strong
Word List A	92.63	83.21
Word List B	84.11	83.35

Table 7.3 Optimisation Step Three Interim ERRT Results

7.2.4) Step 4:

Following the removal of a large number of endings in step three it was important to rank the endings removed with the new rule sets so any changes caused by the modifications to the rule sets are considered in the new rankings. Once the endings had been ranked for the second time a similar process of focussing upon the more commonly used endings was implemented. The following endings were removed during this phase;

aring	*entant*	*entual*	*ifying*	*iteness*	*ward*
bings	*dings*	*gings*	*nings*	*pings*	*tings*

The changes to the ERRT values resulting from these ending removals are presented below.

	Weak Ignored	Weak = Strong
Word List A	91.06	82.70
Word List B	81.76	80.37

Table 7.4 Optimisation Step Four Interim ERRT Results

7.2.5) Step 5:

The next step in the optimisation process focussed upon a single ending, *est*. The ranking of the endings in step 4 had highlighted *est* as being involved in a high proportion of errors per removal. Inspection of the ending's usage revealed that its removal was definitely required in many situations. This was demonstrated by a rise in ERRT values when it was deleted. Further inspection revealed that a large number of the errors occurred when the endings followed an *i*, such as *greediest*. An exception was added to prevent the removal in this instance, but did not improve performance as terms such as greediest were left untouched. The correct stem for the term greediest is *greed*, this fact resulted in the ending *iest* being added to the rule set, with the following affect.

	Weak Ignored	Weak = Strong
Word List A	90.90	81.46
Word List B	76.89	74.08

Table 7.5 Optimisation Step Five Interim ERRT Results

7.2.6) Step 6:

Once the problem with *est* had been solved the ranking of the endings was performed and step 4 was repeated again focussing on the more commonly removed endings. This resulted in the removal of another twelve endings from the rule set presented below, together with the ERRT results for the new ending list.

ents	*ery*	*inate*	*lic*	*th*	*um*
ship	*ships*	*shiped*	*shiper*	*shipers*	*shipful*

	Weak Ignored	Weak = Strong
Word List A	89.28	80.36
Word List B	74.45	72.09

Table 7.6 Optimisation Step Six Interim ERRT Results

7.2.7) Step 7:

The optimisation process has so far focussed on the removal of endings from the rule set which are commonly used. This has the advantage of giving larger improvements in the ERRT values when poorly performing entries are removed. However once the obviously erroneous endings have been removed and the error rate falls below 0.5 it becomes unclear whether removing an ending would be of benefit. For this reason the following step focusses upon endings which although being removed infrequently were always involved in at least one error.

Inspection of the ranked endings gave rise to twenty four candidates for removal. Manual inspection confirmed that the following nineteen endings were causing errors every time they were removed

alion	*alions*	*ature*	*atured*	*atures*	*eal*
enian	*enians*	*entated*	*entful*	*erially*	*erian*
erians	*eship*	*eships*	*mentary*	*ific*	*inism*
ority					

	Weak Ignored	Weak = Strong
Word List A	88.95	78.48
Word List B	73.37	70.87

Table 7.7 Optimisation Step Seven Interim ERRT Results

7.2.8) Step 8:

The final step deals with a problem specific to word list A which contains a number of terms ending either *'s* or *s'*. Although the Lovins stemmer removes both of these from a word, compound endings including them are not included in the ending list. To solve this error an extra step, removing either ending was added to the algorithm. This is performed at the beginning of the stemming phase of the algorithm and resulted in a noticeable improvement in the ERRT values when stemming is performed on word list A

	Weak Ignored	Weak = Strong
Word List A	85.06	73.95
Word List B	73.37	70.87

Table 7.8 Optimisation Step Eight Interim ERRT Results

8) Final Results and Conclusions:

8.1) Results:

The following section will present the final results gathered after all steps of the optimisation process had been completed. The graph below displays the average ERRT values for each optimisation step, with the constant values of the original Lovins, Paice and Porter stemmers displayed. The average is taken from the two word lists and for when both weak barriers are ignored and when weak barriers are treated as strong.

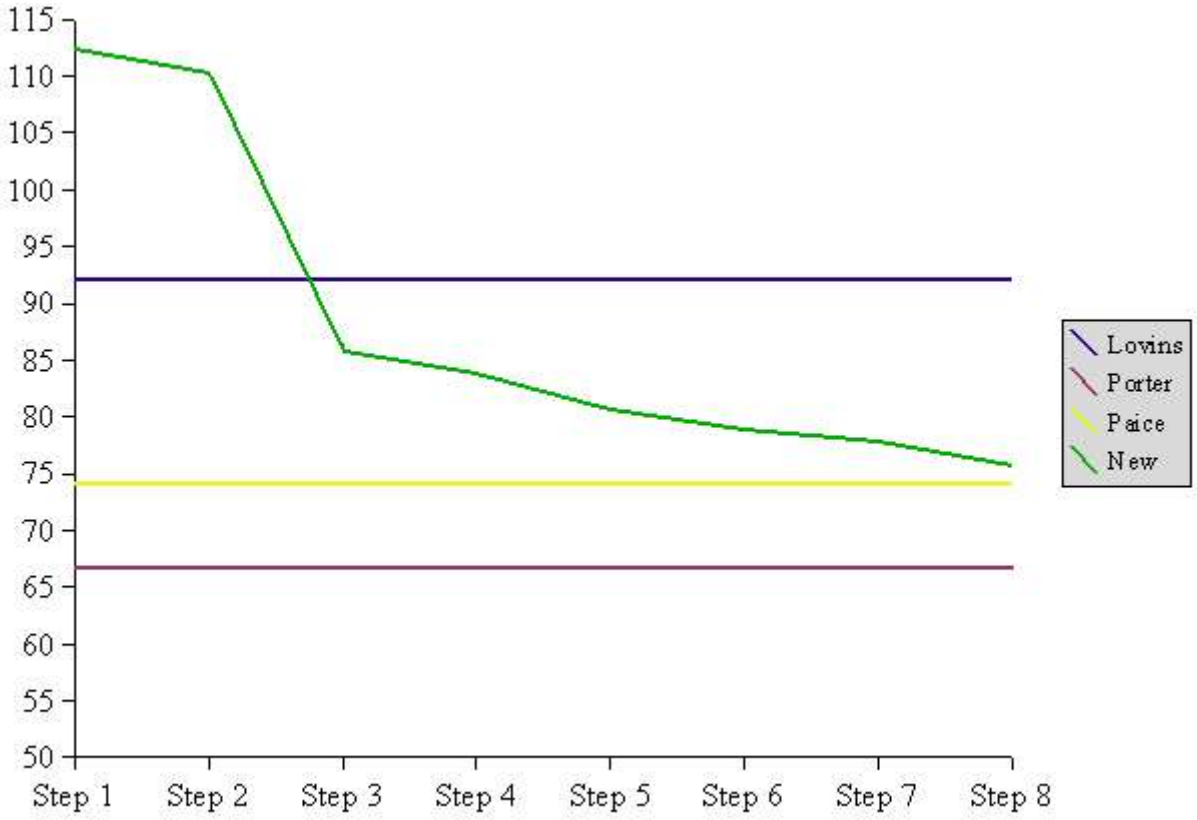

Figure 8.1 Graph of Average ERRT Values During Optimisation Steps

As the graph shows the initial addition of the endings and recodes gathered from the other stemmers caused a large rise in the number of errors in the system. This was easily addressed in step 3 with the removal from the ending list of the endings that were involved in a high proportion of errors and were often removed.

The table below presents the four values used to calculate the average ERRT value for the original three stemmers and the value of the improved Lovins stemmer following all eight optimisation

steps. The table shows that original Lovins stemmer performed the worst in all cases and that the other stemmer's performance varied depending on the word list tested. The exacting ordering of the ERRT values for lists A and B are as follows;

Word List A: *Lovins > New Lovins > Porter > Paice*

Word List B: *Lovins > Paice > New Lovins > Porter*

These ordering indicate that the most consistent of the stemmers is the Porter algorithm. It also displays the impact that different grouped word sets, created by different evaluators can have upon a stemmer's performance.

	Word List A		Word List B	
	Weak Ignored	Weak = Strong	Weak Ignored	Weak = Strong
Lovins	95.64	90.23	91.79	91.47
Paice	71.27	54.26	87.97	83.13
Porter	83.59	67.33	61.66	55.08
New Rules	85.06	73.95	73.37	70.87

Table 8.1 Final ERRT Values

8.2) Conclusions:

Overall the project aimed to develop a method for systematically optimising and refining the rule sets for the Lovins stemmer. The work built upon previous research that provided a tool for evaluating stemming algorithms and extended and updated the tool to allow for more effective research. The optimisation method presented here relies upon the implementations of the stemmers to retain all the information about the decisions that are made and actions taken when conflating a word. This raw data from the stemmers can then be processed to provide information to the evaluator indicating which elements of the rule set are performing poorly, and what modifications would allow for the greatest improvement in stemmer accuracy and performance.

The results presented in table 8.1 show that the original Lovins stemmer consistently performs worse than the Porter or Paice stemmers. Once the optimisations had been performed the performance not only improved considerably from the original stemmer but also performed better than the Paice stemmer in two of the four results, although not quite being able to produce a better average that either Paice or Porter.

The table also displays the large variations in ERRT values that occur when using different word lists, and even within word lists significant differences can be noted between ignoring weak barriers and treating them as strong. These variations in ERRT values demonstrates the impact different word lists, taken from distinct domains, grouped by various users have on stemmer performance. To

improve the validity of any future results, and to assist in verifying any improvements gathered whilst optimising the rule sets, continued expansion of the input data needs to be performed. This expansion must include the creation of new grouped word files from word lists taken from differing subject domains. These grouped word lists must also be larger than those presented here, and be grouped by as larger group of evaluators as possible.

It can be seen in graph 8.1 that although a large improvement in the ERRT values for the new rules was obtained during the first ranking and removal of the endings after this the improvements were consistent but more modest. These diminishing returns are to be expected as the errors within the algorithm become more dispersed. Increases in performance also become more difficult to obtain as although removing an ending from the rule set might improve the ERRT values slightly, unless it is a clear improvement it cannot be verified that removing the ending will mean a consistent improvement across all groups of words.

8.3) Future Work:

The method and tools for improving the rule sets of the Lovins stemmer has been presented and proven to increase the accuracy of the stemmer. The technique involves a systematic investigation and refinement of the stemmer which could be continued indefinitely, although with increasingly diminishing results.

One possible improvement of the evaluation suite is the incorporation of the Optimisation package into the GUI allowing for easier and faster implementation of the optimisation steps. The generalisation of the optimisation package to allow the handling and ranking of endings/rules from the Paice and Porter stemmers could allow the same steps to be applied and would open up the possibility for similar enhancements to these algorithms

The need for more and larger grouped word files is constant, but the time and effort required for their creation can be prohibitive in obtaining large sets of results. A method based on utilising a *constant vocabulary* approach with a thesaurus to place words into multiple concept groups is presented in [21]. This method would reduce the work required by the evaluator in creating grouped words files but does create a problem. The use of an automated method for the creation of grouped word sets that are used to evaluate automated conflation techniques such as stemming algorithms based on error counting implies that the original method's decisions are assumed to be all correct. If the method used to create the grouped files does provide more accurate results then the advantage of implementing stemming algorithms becomes increased efficiency in matching terms.

The time taken to create the grouped word set used in this research was about ten percent of the total project duration. Although this is not considered prohibitive given the results obtained other

approaches to word grouping need to be considered if they offer significant improvements in the time to create grouped files. Once such approach is to examine words beginning with the same few first letters, and grouping together those that tend occur in close proximity in very large word samples of texts, e.g. those occurring within the same web page.

Bibliography:

[1] Dawson, J.L. (1974) *Suffix Removal for Word Conflation*, Bulletin of the Association for Literary and Linguistic Computing, 2(3): 33-46

[2] Frakes, W.B. (1984) *Term Conflation for Information Retrieval*, Cambridge University Press

[3] Frakes, W.B. & Baeza-Yates, R (1992) *Information Retrieval: Data Structures and Algorithms*, Englewood Cliffs, NJ, Prentice Hall

[4] Frakes, W.B. & Fox, C.J. (2003) *Strength and Similarity of Affix Removal Stemming Algorithms*, SIGIR Forum, 37: 26-30

[5] Hooper, R.P. (2004) *Development and Evaluation of a Stemming Algorithm*, Unpublished

[6] Hull, D.A. (1996) *Stemming Algorithms – A Case Study for Detailed Evaluation"*, JASIS, 47(1): 70-84

[7] Hull, D.A. & Grefenstette, G. (1996) *A Detailed Analysis of English Stemming Algorithms*, Xerox Technical Report

[8] Kowalski, G. (1997) *Information Retrieval Systems, Theory and Implementation*, Kluwer Academic Publishers

[9] Krovetz, R. (1993) *Viewing Morphology as an Inference Process*, In Proceedings of ACM-SIGIR93, pp191-203

[10] Lennon, M., Pierce, D.S., Tarry, B.D. & Willett, P. (1981) *An Evaluation of some Conflation Algorithms for Information Retrieval*, Journal of Information Science, 3: 177-183

[11] Lovins, J. (1968) *Development of a Stemming Algorithm*, Mechanical Translation and Computational Linguistics, 11: 22-31

[12] Lovins, J. (1971) *Error Evaluation for Stemming Algorithms as Clustering Algorithms*, JASIS, 22: 28-40

[13] O'Neill, C. (2001) *Development and Evaluation of Stemming Algorithms*, Unpublished

[14] Paice, C.D. (1990) *Another Stemmer*, SIGIR Forum, 24: 56-61

[15] Paice, C.D. (1977) *Information Retrieval and the Computer*, Macdonald and Jane's, London

[16] Paice, C.D. (1996) *Method for Evaluation of Stemming Algorithms based on Error Counting*, JASIS, 47(8): 632-649

[17] Paice, C.D & Black, W.J. (2003) *A Three-pronged Approach to the Extraction of Key Terms and Semantic Roles*, In Proceedings of RANLP 2003, pp 357-363

[18] Pearsall, J. (2001) *Concise Oxford English Dictionary, Tenth Edition, Revised*, Oxford University Press

[19] Porter, M.F. (1980) *An Algorithm for Suffix Stripping*, Program, 14(3): 130-137

[20] Salton, G. & McGill, M.J. (1983) *Introduction to Modern Information Retrieval*, McGraw-Hill, New York

[21] Sanchez, R. M (2005) *A Generalization of the Method for Evaluation of Stemming Algorithms Based on Error Counting*, Proceedings of SPIRE 2005 (to be published)

[W1] Wikipedia, The Free Encyclopaedia.
http://en.wikipedia.org/wiki/Main_Page

[W2] The Porter Stemming Algorithm.
http://www.tartarus.org/~martin/PorterStemmer

[W3] The Lancaster Stemming Algorithm
http://www.comp.lancs.ac.uk/computing/research/stemming/

[W4] The Lovins Stemming Algorithm
http://snowball.tartarus.org/algorithms/lovins/stemmer.html