INTELLIGENT SEARCH TECHNIQUES FOR LARGE SOFTWARE SYSTEMS

Huixiang Liu

Thesis

submitted to the Faculty of Graduate and Postdoctoral Studies
in partial fulfilment of the requirements
for the degree of Master of Computer Science

August 31, 2001

Ottawa-Carleton Institute for Computer Science
School of Information Technology and Engineering
University of Ottawa
Ottawa, Ontario, Canada

ACKNOWLEDGEMENTS

I would like to acknowledge the help that I have received during my research.

Grateful thanks to:

- Dr. Timothy Lethbridge, my supervisor, for his support, guidance, patience and intelligent comments.
- The KBRE group for their help, comments, and the valuable discussions with them.
- The software engineers who participated in this study.
- My friends for their concerns and encouragements.
- My family, for the endless support to me.

ABSTRACT

There are many tools available today to help software engineers search in source code systems. It is often the case, however, that there is a gap between what people really want to find and the actual query strings they specify. This is because a concept in a software system may be represented by many different terms, while the same term may have different meanings in different places. Therefore, software engineers often have to guess as they specify a search, and often have to repeatedly search before finding what they want.

To alleviate the search problem, this thesis describes a study of what we call *intelligent search techniques* as implemented in a software exploration environment, whose purpose is to facilitate software maintenance. We propose to utilize some information retrieval techniques to automatically apply transformations to the query strings. The thesis first introduces the intelligent search techniques used in our study, including abbreviation concatenation and abbreviation expansion. Then it describes in detail the rating algorithms used to evaluate the query results' similarity to the original query strings. Next, we describe a series of experiments we conducted to assess the effectiveness of both the intelligent search methods and our rating algorithms. Finally, we describe how we use the analysis of the experimental results to recommend an effective combination of searching techniques for software maintenance, as well as to guide our future research.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1. INTRODUCTION

1.1 CURRENT PROBLEMS OF SEARCHING IN LARGE SOFTWARE SYSTEMS

Many researchers in the reverse engineering community have shown that search is a major task of software maintenance [14]. In large software systems, searching in the source code is even more time consuming, so the task of program comprehension is even more difficult. Therefore, tools targeted to the search problems in source code should help improve the effectiveness of program comprehension. Search tools, such as Unix 'grep' or the facilities provided with code exploration systems such as Source Navigator [25], are used every day by many software engineers to facilitate program comprehension.

Even with the regular search tools, however, Lethbridge *et al.* [14] have found that when working with source code, search can account for as much as 28% of the individual activities performed by software engineers. In particular, it takes them a long time to find the files that include a certain string (e.g. using grep), and then to look for the string in the files found (e.g. using an editor).

The time spent on search is partly due to the fact that software engineers have to search for the same string over and over again because they can not remember or access the results of previous queries. Some of the source code exploration tools such as Source Navigator [25] and TkSee [13] partially solve this problem by remembering the recent query history for the users. Consequently, software engineers can work on the same result list repeatedly without performing the same query again.

Another factor that contributes to the massive amount of time and effort spent on search is the synonym and polysemy phenomenon [16] in software systems. The *synonym phenomenon* refers to the fact that the same concept can be represented in many different ways. *Polysemy* means a word can have different meanings in different contexts. Therefore, it takes a lot of time for people to figure out what words have been used by other programmers to represent a certain entity[1].

[1] By *entity*, we are referring to software units that represent concrete objects or abstract concepts in software systems. In source code, entities could be files, routines, variables, constants and other identifiers.

Moreover, even if people know exactly which words represent a concept in a software system, they might not know in which format the words appear. For instance, some programmers use “msg” to refer to “message”, while some others might use “mssg” or “mes” instead. Consequently, when a user searches for something with the name indicating that it is a message, he has to try all the possible forms of “message” before he can get a comprehensive list of results.

The existence of synonyms, polysemy, and multi-morphology of words makes searching in source code systems much harder because it is difficult to find out exactly what names are used to represent certain concepts or objects in the system, as well as in which format the names appear. As a consequence, the software engineer has to figure out a possible query string, perform the search, and investigate the results. If none of the results is the expected target, the software engineer must come up with the next query, and start the search process again. This repetitive procedure is necessary for not only novices of the systems, who have to acquire knowledge of the applications that are new to them, but also for experts, who, however long they have been working with the systems, can not remember all the details.

Unfortunately, normal search only retrieves results by exact string matching. Sometimes such exact matching can not find anything. For example, if a user is interested in information related to the query, "call forward while busy", normal search facilities might find a few comments in source code that happen to contain exactly the same character string, but can not return an entity whose name is "callForwardBusy", or "cfb". Some other times, although exact matching does find several results, the users expect more. For example, if a user is working with "listdbg" and hopes to learn more about it by searching for relevant entities, normal search will only help retrieving "*listdbg*"-like names (the * symbol in this thesis represents zero or more substitutions of any character), but will miss "list_debug", if it exists.

Clearly, the gap between the users’ information need and the actual query strings they specify is a very important issue to address. We need advanced techniques to address this problem, and to retrieve the entities in large software systems that satisfy users' information needs.

In the remainder of this chapter we will compare related searching techniques. In the subsequent chapters, we will present our solution to the searching problem of large software systems.

1.2 RELATED RESEARCH

In the past few decades, many search tools and software information systems [2] have been developed to assist software engineers in working with the large collection of knowledge in source code. We can classify the search techniques of the tools into two categories, data retrieval and information retrieval.

1.2.1 Data retrieval

Baeza-Yates defined *Data Retrieval* as:

> *the retrieval of items (tuples, objects, Web pages, documents) whose contents satisfy the condition specified in a (regular expression like) user query*. [5]

Data retrieval processes the query strictly according to the user's specification. In other words, these tools assume that users know precisely what they want, and their information need is accurately expressed by the query string. Regular expression search is a member of this category because it does nothing to interpret users' information requirements. Therefore, a document is judged as either a hundred percent conforming to the requirement, or not conforming at all. Every retrieved result is considered "relevant" to the query, and no evaluation of the result's pertinence is necessary. Hence, any single item in the result list that is not consistent with the search condition can only be attributed to a defect in the data retrieval mechanism.

Unix 'grep' and most of the commercial tools [15] that support search functions such as Source Navigator [25] and Microsoft Visual C++ [19] belong to this family. There are also some software visualization tools [15], such as ShriMP [23] and TkSee [13], in academia that allow searching for the entities in source code. All these tools provide exact string matching and pattern matching facilities such as regular expression search.

To improve search efficiency, some of these tools [12] employ DBMS-like mechanisms to support data retrieval in large software systems; this is because such mechanisms are very efficient for fully structured or semi-structured documents like source code. These tools first extract and store in a database the entities defined or used in the system. Then, during the search processes, the DBMS query mechanism can quickly retrieve from the database the entities that match the search conditions, and bring the searcher to the location of the entities in the system.

1.2.2 Information retrieval

Unlike data retrieval, information retrieval focuses on the information need represented by the query. *Information Retrieval*, as Baeza-Yates defines it, is:

> *part of computer science which studies the retrieval of information (not data) from a collection of written documents. The retrieved documents aim at satisfying a user information need usually expressed in natural language.* [5]

Information retrieval assumes that users might not have provided accurate descriptions of their information need; hence the techniques automatically expand the queries by looking for relevant or useful information represented by partial matching, as well as exact matching. Unlike in data retrieval, a certain fraction of inaccurate items among the result set is tolerable in information retrieval because of the ambiguity of natural language.

There have been some applications of information retrieval technology in software information systems [2]. These applications include the CATALOG system of AT&T Bell Labs [10], the IBM Share System [7], the British ECLIPSE [27] project, the European PRACTITIONER project [6], and recently IBM's jCentral [30, 31], to name a few. No matter what techniques these studies have used, they have provided us many valuable experiences of software information retrieval systems.

1.3 LIMITATIONS OF PREVIOUS RESEARCH

Unfortunately, previous practices of data retrieval and information retrieval in software systems have not solved the problem we described in section 1.1. Data retrieval techniques are, of course, not enough because users' information needs are not their utmost concern. However, few of the previous information retrieval studies has addressed our problem, either.

It is notable that most of the previous software information systems using information retrieval techniques were designed for software reuse. What these systems had in common was that they had a collection of reusable software components represented with some predicates to describe their functionality. The software components were then classified and indexed according to the predicates. During the retrieval, the systems would match user queries with the predicates, the representations of reusable software, and hence locate the software for the user.

For example, CATALOG consisted of a small collection of reusable program modules. In order to help retrieve these modules, it first prepared and stored the textual descriptions of them, and then it constructed the index using every single word of the descriptions. During the retrieval, CATALOG compared the queries with the indexed terms, and returned some relevant descriptions. These descriptions would further point to the corresponding reusable modules.

Hence, most of these software information systems either did not deal with source code directly, or focused mainly on bigger software components, such as functions, procedures, packages and programs. Moreover, if they were automatic classification and indexing systems, their index terms were mainly extracted from the development documents and/or comments in source code that were more similar to normal unstructured text documents written in natural language than source code.

IBM’s jCentral [31] is a tool that is aimed at more than software reuse and deals with source code directly. It searches and locates in the Web space Java resources, including source code, news group articles, FAQs, and so on. It generates and indices a summary describing the features of each resource from descriptive text, e.g. the text on the HTML page that contains the source code, as well as extracts the Java-specific features, such as class, method names and signature, properties and etc., if the resource is Java code.

Unfortunately, the documents we have found about jCentral do not indicate whether it further analyzes the information embedded in names of classes and methods, which as discussed below are one of the most important pools of information for software maintenance.

1.4 DIFFICULTY OF SEARCHING SOFTWARE ENTITIES

In the area of software maintenance, the search problem requires much finer granularity, i.e. the names of files, routines and other identifiers and even the sub-components of the names, because software maintenance mostly has to involve source code, and hence involves searching entity names in source code. As in one of our previous examples, a software engineer might want to learn more about how "call forward while busy" is handled in the system he is maintaining. Thus the results of the retrieval should be some entities related to the query subject, rather than several reusable modules that process the "call forward while busy" feature.

In addition, entity names usually encode lots of information about the entity. Sometimes software engineers might use multiple words to convey such information. For instance, a variable named "dpn_nsi_call_forward_busy" implies that it has a strong relationship with the above query. Nevertheless, in order to learn such a relationship, we have to extract the information from the names first. Yet as we will discuss in detail later, most of the names of files, routines, identifiers etc. contain *ad hoc* abbreviations and other transformations of names of the concepts they represent, and hence are very complex to deal with.

Few studies have analyzed the composition of entity names in software systems. Although Anquetil *et al.* [3] have introduced methods of splitting file names for file clustering, those methods have not been used for information retrieval in software systems.

Moreover, current information retrieval techniques employed in normal systems are mainly based on words. Not all of these techniques are efficient or suitable for software systems with many abbreviations. Therefore, we need to investigate the techniques that are aimed at the information retrieval problems of software systems.

1.5 MOTIVATION AND OBJECTIVES

According to the search problems in software maintenance, as well as the limitations of previous studies in information technology, we need to investigate how information retrieval techniques can be specifically adapted to the requirements of program comprehension.

Also we want to build a new search tool that employs advanced algorithms. The tool will accept free-style natural language user queries as the input, perform query expansion using the information retrieval techniques, and return the retrieved entities to the user. Therefore, when normal search tools fail to find enough results, the new tool will give both novices and experts of large software systems additional search power. For novices, the tool will facilitate their learning process of a new system. For experts who have been working with the systems for a long time, the tool will reveal many aspects that they might not be familiar with.

1.6 CONTRIBUTIONS OF THE THESIS

Our solution to the search problem in large software system is to automatically expand the query in order to find more relevant entities. Query expansion is achieved by transforming the query string using the text operation methods in information retrieval. We select a set of information retrieval techniques to perform the transformation, and incorporate the techniques into current search facilities. We have developed an *intelligent search* tool to implement our ideas and to assess the effectiveness of the selected information retrieval methods. We also show through a series of experiments that these techniques will increase the search power for both novices and experts of software maintenance. Therefore, the major contributions of this study are as follows:

1. We have proposed a general model of applying information retrieval techniques to software maintenance. We have implemented and integrated an intelligent search tool into the existing source code browsing tool “TkSee” [12, 13] such that the new search facility benefits from both the efficiency of data retrieval techniques and the flexibility of information retrieval techniques.

2. In the model, we have adapted some information retrieval techniques to software systems, i.e. most of the text operation methods such as truncation, stopword removal, techniques based on dictionary or knowledge-base lookup, and so on. We have also developed *abbreviation concatenation* and *abbreviation expansion* methods specifically for software maintenance to handle the large number of abbreviations in software systems.
3. Similar to other information retrieval systems, the ranking algorithm is the kernel of our model. We have designed the rating algorithms specifically for software systems, and have used the statistical output of the algorithms to analyse the performance of the search techniques and to recommend better combinations of search strategies.
4. We also have evaluated the accuracy of our rating algorithms through human experiments. Their relative high accuracy, as well as their ease of implementation, verifies their value in large software systems.

In general, this thesis will be of interest to researchers of information retrieval and program comprehension. In addition, it will be of interest to researchers and software engineers who would like to study the decomposition of the abbreviations in source code.

1.7 THE ORGANIZATION

Chapter 2 proposes the general architecture of information retrieval for large software systems, and highlights its differences as compared to normal information retrieval models.

Chapter 3 explains the selection of information retrieval techniques based on the results of our pilot experiments.

Chapter 4 describes in detail how the selected information retrieval techniques are used in our intelligent search tool.

Chapter 5 focuses on our rating algorithms.

Chapter 6 presents a series of experiment results to assess the effectiveness of the search techniques and the rating algorithms.

Chapter 7 sums up our research and points out future study directions.

CHAPTER 2. ARCHITECTURE OF THE INTELLIGENT SEARCH TOOL

This chapter first introduces the generic model of information retrieval systems. Then, according to the requirements of software maintenance, we propose a model for using information retrieval technology in source code systems. We also discuss the difference between this model, on which our intelligent search tool is based, and the generic model. Finally, we present how our intelligent search tool will work under the new structure.

2.1 GENERIC MODEL OF INFORMATION RETRIEVAL

Information retrieval techniques were originally designed to solve the problems of searching in a large collection of unstructured text documents written in natural language such as books and articles in a library. Now, these techniques are also extensively used in the search engines of the Web.

As Baeza-Yates [5] has said, information retrieval allows imprecise interpretation of query requirements because of the ambiguity of natural language. Therefore, information retrieval usually needs processes to "interpret" the query, to retrieve the expanded query condition according to the interpretation, and to evaluate the closeness of the result to the original query. Figure 2.1 shows the generic model of information retrieval. The left side of the model shows the above three steps of the retrieval process, while the right side is the pre-processing phase, which is necessary before any retrieval process can be initiated. One advantage of information retrieval techniques for large document systems is that the cost is relatively low because both the pre-processing phase, including the index construction step, and the query phase can be fully automated and efficiently.

2.1.1 Pre-processing

At the right-hand side of the information retrieval model in Figure 2.1, a collection of text documents are pre-processed by some automatic text operation methods, such as stopwords removing, stemming, and etc. The result is a collection of words, which is

considered as the logical view [5] of the document. Sometimes the logical view can also be some keywords manually assigned by the authors or the librarians.

Next, the indexing phase uses the words describing the logical views of the documents to construct a word-oriented index. One of the most popular indexes of this type is called *inverted file* [5]. In an inverted file, each word points to a list of all the documents in which it occurs. An index like inverted files is a very important structure in information retrieval because it, "allows fast searching over large volumes of data" [5].

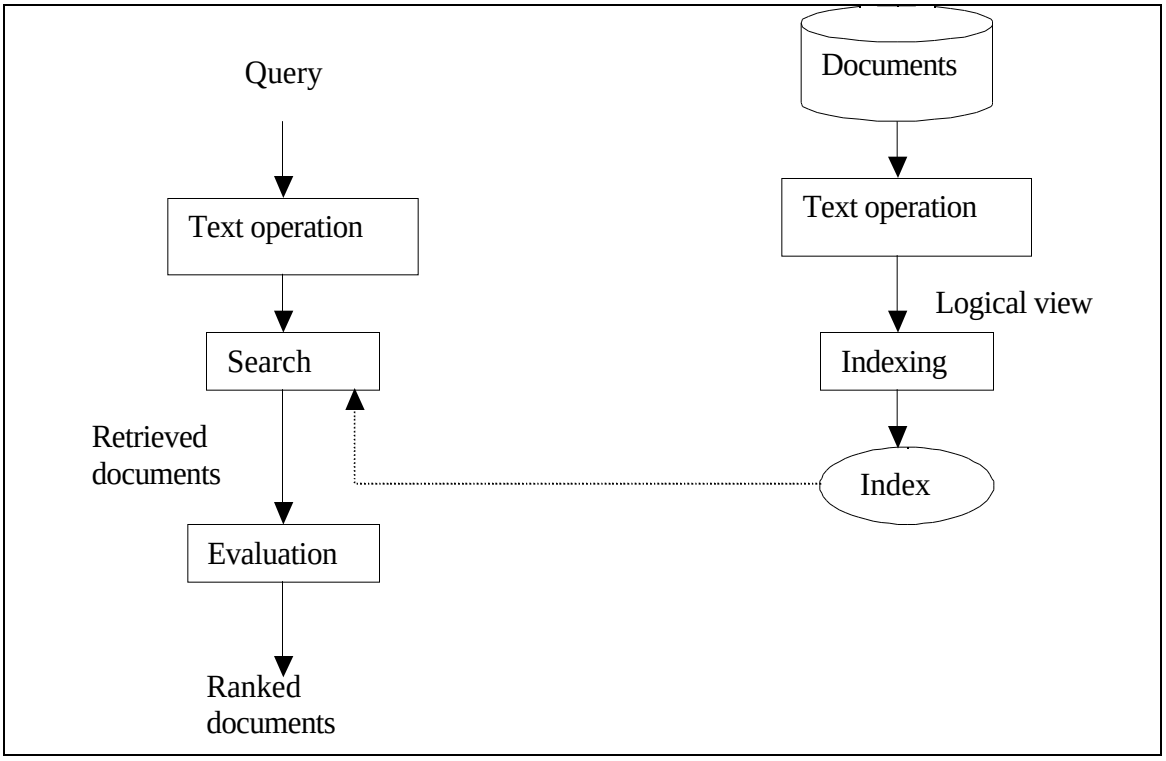

Figure 2.1 A model of normal information retrieval systems

2.1.2 Retrieval

During the query process, which is shown at the left-hand side of the model, the information retrieval tool first performs similar text operations on the user query as on the original documents during pre-processing. The text operations are the major methods in this

model used to interpret users' information needs, and are one of the major differences between information retrieval and data retrieval, where there is no logical operation on the original query before the search. The output of the text operation is a list of words, which is the internal representation of the user's information need.

During the search phase, each of the words obtained from text operation is used to locate, through the index, a list of all the documents in which it occurs. When multiple words are present in the query, search returns the union of the documents retrieved by all the words. In short, searching is a process of matching the words in the documents with those in the query.

Lastly, every retrieved document is evaluated by its relevance to the query. Usually this evaluation is dependent on a ranking algorithm that computes a real number for each document. A document with a larger number is considered more likely to be relevant. Next, the retrieved documents are returned in descending order according to the results of the ranking algorithm; therefore, users have a better chance to inspect the most relevant documents, which are at the top of the ordering, than the irrelevant ones. As a result, the ranking algorithm is regarded as an essential part of an information retrieval system.

2.2 THE DIFFERENCES OF INFORMATION RETRIEVAL BETWEEN SOURCE CODE AND REGULAR DOCUMENTS

2.2.1 The difference between source code and regular documents

The following are two key features of source code that distinguish it from ordinary documents. Understanding these features will help us decide what methods to apply in software information retrieval systems.

- Source code files are not ordinary English-language documents, even though their programming languages are English-based. The grammar is that of programming languages, not English. After removing the reserved words from the source code, the remaining words are names, values, or comments.
- Names in software systems are different from the names we use in other types of literature. Even though most software systems involve some naming conventions,

the names can be anything from a single letter, or a single word, to a combination of words; and the words are frequently abbreviated.

Therefore, it depends highly on the programmer which words to choose to represent an object in a software system, and in which format the words appear in the object's name. In some cases, a name can be just a concatenation of letters so that nobody other than the original programmer can possibly judge its meaning.

2.2.2 The inappropriateness of the normal information retrieval model in large software systems

The differences between source code and conventional text documents determine the difficulty of applying the regular information retrieval model in large software systems. The major difficulties lie in the following aspects of the model:

- **Target of retrieval**: Although "documents" in regular information retrieval refer to various types of text units [5], they typically contain a set of individual words. The purpose of the queries, therefore, is to locate the documents containing certain words. However, in software systems, programmers usually look for the names of the entities, e.g. "listdbg", that are pertinent to what is specified in the query, rather than the bigger units such as a chunk of code. The direct target of the search, therefore, is the names, which can then bring the users to the position where the entities are defined or used in the system. Consequently, names are a special type of mini-documents, in which component units might not be clearly separated as are words in regular documents.
- **Text operations**: The lack of word delimiters, as well as the use of abbreviations in the names, lead to the situation that the text operations used in normal document systems might not be appropriate or adequate in source code systems. Especially, as discussed in detail in the next chapter, there are no methods proposed in information retrieval to identify and separate the concatenated abbreviations in the names of

entities. Hence, we need to select and design an efficient and suitable set of text operations for source code systems.

- **Indexing**: The biggest problem lies in the construction and maintenance of the index. In normal systems, it is possible to automatically build a word-oriented index in a relatively short time, and this index is critical for information retrieval in these word-oriented systems. In software systems, as previously mentioned, we are looking for entity names, which might not be word-oriented, but contain components of various abbreviated formats. Thus, in order to efficiently retrieve entity names based on a few pieces of information from the query string, we need to identify all components of the names, and store and index all the unique component formats that have ever appeared in the systems. However, this index is different from the index of entity names, which can be automatically constructed by parsing the source code as we have done in TkSee. The difficulty of decomposing names containing contracted abbreviations determines that it is very hard to automatically identify the components and index all the unique ones. Sometimes it is even hard for experts of a system to split the names correctly. Therefore, the cost of index construction and maintenance is very high. It is impractical to automatically or manually build a complete index for the components of the names in a large software system.
- **Evaluation**: Similar to text operations, most of the evaluation methods in conventional systems are word-oriented, which is inadequate to solve the problems in software systems. The existence of abbreviations correspondingly call for variants of existing rating algorithms.

2.3 PROPOSED MODEL OF INFORMATION RETRIEVAL FOR SOFTWARE MAINTENANCE

However different they are, source code and regular documents are similar in that terms appearing in them are vague. Therefore, information retrieval systems aimed at software engineers' information needs also require the three basic steps of normal information retrieval systems, i.e. interpretation of the query, retrieval, and evaluation of the results. Based on this similarity, as well as the differences reviewed above, we propose a model of

applying information retrieval in software maintenance, shown in Figure 2.2. This is a model that combines both data retrieval and information retrieval techniques.

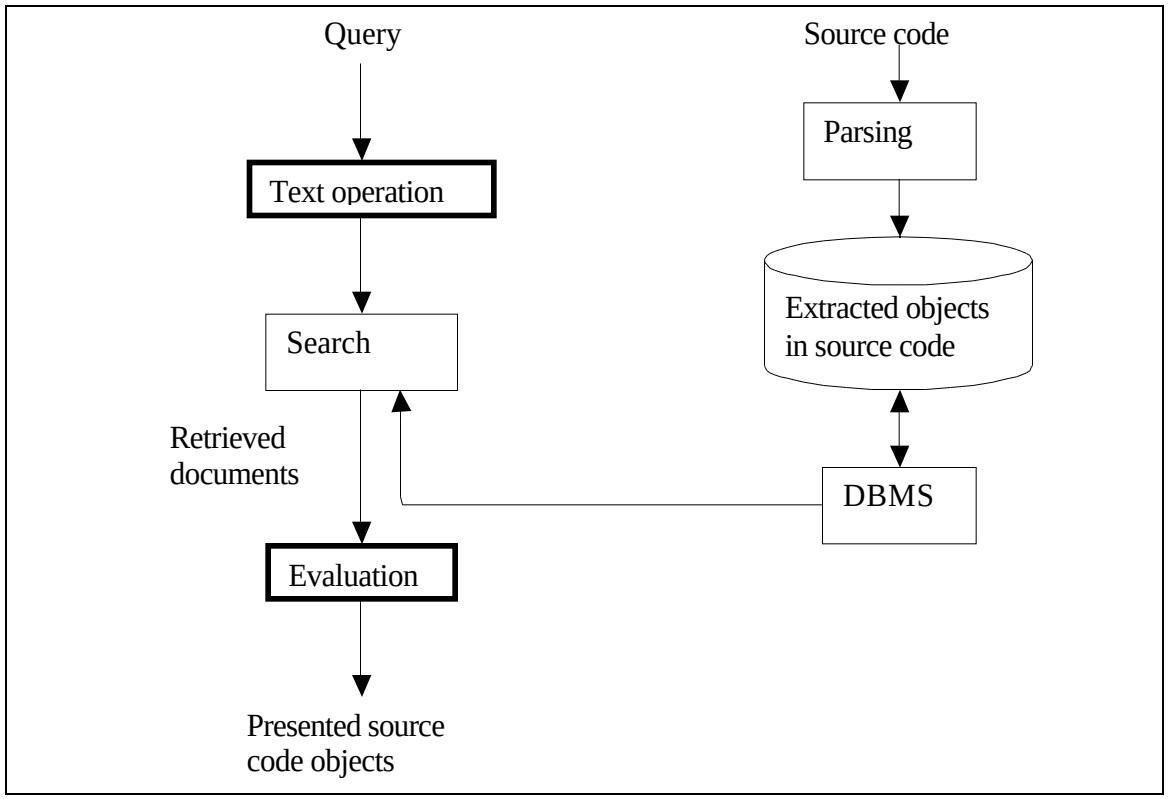

Figure 2.2 The model of software information retrieval systems

2.3.1 Data retrieval

If we remove the two highlighted procedures ("Text operation" and "Evaluation") in Figure 2.2, we obtain a typical data retrieval model, which has been implemented in many source code browsing tools such as TkSee [12]. The search facility fetches from the database the objects that match the query, and returns them to the user. Powered by some very efficient mechanisms such as DBMS, this process is capable of handling a large volume of data (millions or even billions of records) in a short time.

Comparing our model with the model in Figure 2.1, the major difference resides in the right side, i.e. the pre-processing phase. There is no indexing procedure during the pre-

processing phase in our model, and no index for information retrieval is constructed. Although the database might contain a table that serves as an index of entities in the system, the index is not the one we need for information retrieval purpose. The reason, as have been previously discussed, is related to the impracticality and the high cost of constructing and maintaining the index of the components in entity names. Without the index, the information retrieval facilities may lose the high efficiency of normal systems. However, our experiments will show later that, powered by the data retrieval mechanisms, our model conveys the advantages of information retrieval with acceptable performance.

2.3.2 Information retrieval

Built on top of an existing data retrieval infrastructure, our information retrieval model includes the same three procedures during retrieval as those in conventional systems. However, these procedures function in a different fashion from those in Figure 2.1 due to the difficulties mentioned in section 2.2.

Primarily, text operations and result evaluation methods have to be designed specifically for software systems characterized by the prevalent usage of abbreviations; these methods are the focus of this thesis.

The search process is also different from the conventional ones because it is built upon an existing DBMS structure, rather than upon an index of terms. It sends, one by one, every expanded version of the queries to the DBMS, and joins the results of all the retrievals for the next step, the evaluation stage. The expanded queries are in general regular expressions, which can be handled by the DBMS mechanism.

2.4 THE INTELLIGENT SEARCH TOOL

Based on our model, we have built an intelligent search tool that employs what we call *intelligent search techniques* as well as some pre-existing facilities. The tool utilizes the query module of TkSee [12], a source code browsing tool, as the underlying data retrieval mechanism. Moreover, it makes use of some knowledge bases and dictionaries to support information retrieval. In this section, we first describe the existing facilities in the

"environment" part, and then introduce the overall intelligent search procedure, which will be the focus of the following chapters.

2.4.1 The environment

We have a long-term relationship with a large telecommunication company, and have conducted reverse engineering studies on one of their legacy systems that is over 17 years old. The system, which consists of about 1.5 million lines of code, is a mixture of programming languages, including C, Pascal and Assembly Language.

In order to help the software engineers to understand the system, our previous studies have developed a source code browsing tool called TkSee [12]. Behind TkSee are a database that serves as an index of all source code elements in a system, and a query mechanism that offers fast grep-like searches of code using the database. Our intelligent search tool generates expanded queries in the form of regular expressions that will be passed to this existing query mechanism.

Our team's previous studies have also constructed a knowledge base system to represent some of the domain concepts used in the legacy system [22]. The knowledge base organizes the concepts in an inheritance hierarchy. When performing intelligent search, we look in this knowledge base for the synonyms, superconcepts, and subconcepts of the original query term.

Another knowledge base we use is a dictionary of abbreviations gathered from the legacy system. The dictionary, which was generated several years ago by the software engineers working in that system, is by no means comprehensive or up-to-date. However, it still provides lots of valuable information when we try to decompose the concatenated abbreviations.

Finally, we obtained from the Web [1] a library of stopwords so that common words can be handled differently.

2.4.2 Intelligent search procedure

For each search request, there are three stages before the presentation of the query results. As shown in Figure 2.3, they are: 1) generating search candidates using several

different algorithms, 2) performing the queries, and 3) evaluating the results in order to choose which ones to present to the user, and in what order.

In the candidate generation stage, we use the text operations in our candidate generation algorithms to compute for the original query a group of additional search candidates. Each of the search candidates is a regular expression that is processed by the backend query mechanism. In addition, this stage also generates a set of evaluation strings from the query for later use in the evaluation stage. As we will present in detail, the use of evaluation strings is at the center of the evaluation.

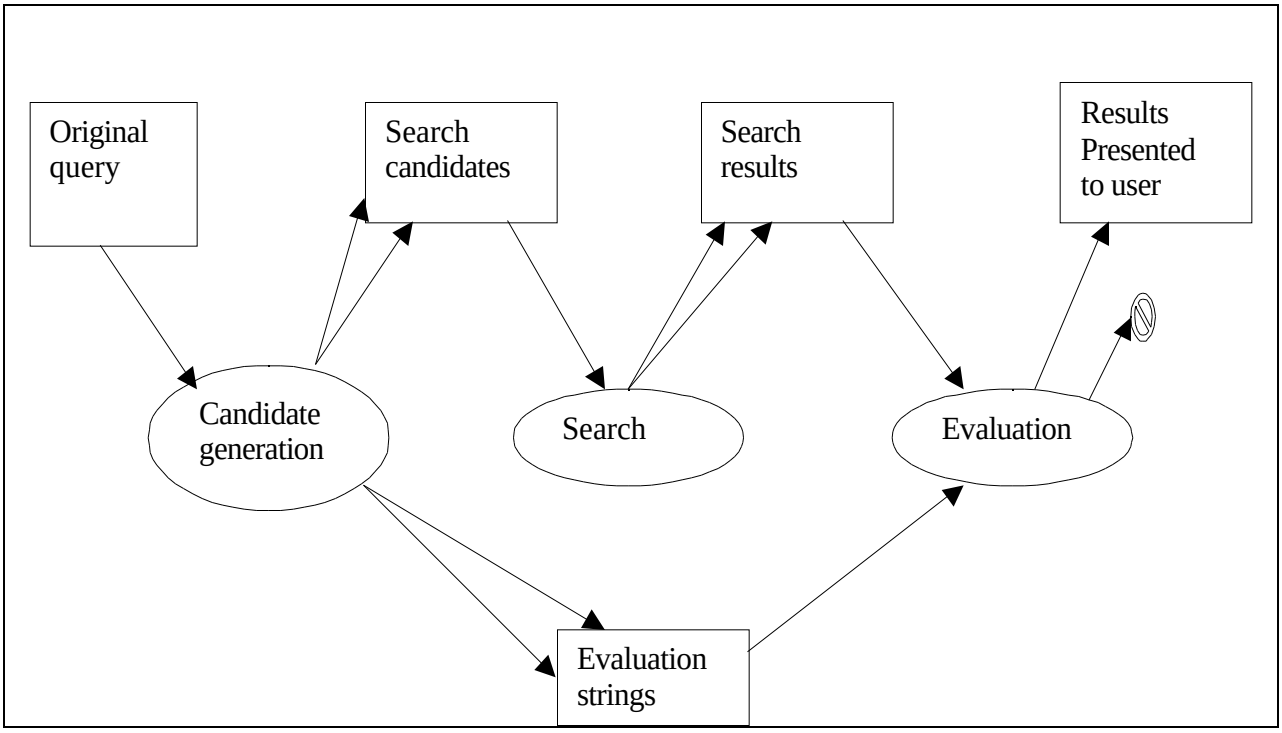

Figure 2.3 Intelligent search procedure

In the search stage, the new search candidates are passed to the query mechanism of TkSee one by one, and each of these candidates will help retrieve a set of search results (entities in software systems) from the database. As soon as all candidates have been queried, their results are merged to remove duplicate findings.

In the evaluation stage, our rating algorithms assess each result by comparing the entity name with the evaluation strings. Then, the assessment will assign a real number, called rating, to each result. According to their ratings, the results are either discarded as non-relevant results, or presented to the user in the descending order of ratings.

Therefore in this procedure, each original query can generate a group of additional search candidates; each candidate may find a group of results later. Conversely, a given result may be found by several different search candidates; each candidate may be generated by different algorithms.

2.5 SUMMARY

In this chapter we have discussed the conventional information retrieval model and disadvantages of using it in source code systems; we have proposed a revised model that benefits software maintenance by employing both data retrieval and information retrieval technologies. We have also introduced the background and the major procedures of our intelligent search tool based on this new model. In the following chapters, we will describe in depth the procedures of the tool: Chapter 3 describes our selection of the text operations for software maintenance; Chapter 4 explains how we incorporate these selected methods into our candidate generation algorithms; Chapter 5 emphasizes our rating algorithms using evaluation strings.

CHAPTER 3. SELECTION OF TEXT OPERATIONS

In this chapter, we analyze conventional text operations in information retrieval for query expansion. In order to learn the usefulness of these methods in software systems, we conducted a small experiment in the large software system we are working with. According to the experiment data, we select a suitable set of methods for software systems. This set of methods, which is called intelligent search techniques by us, includes some of the conventional text operations, plus the supplementary methods we developed particularly for source code systems.

3.1 REGULAR TEXT OPERATIONS

In addition to exact string matching and regular expression matching, many methods have been developed to expand queries in order to generate more search candidates and retrieve more relevant results. Typically, these query expansion techniques generate additional search candidates in the following ways [4]:

- Transforming upper case to lower case.
- Removing numbers and special symbols.
- Removing common words using a list of stopwords.
- Generating acronyms: Taking all the initials from the original string.
- Truncating: Reducing the word's length by removing its final characters.
- Stemming: Removing suffixes and/or prefixes.
- Transforming to spelling variants by using soundex-like methods [24].

The above methods tend to find words that are *morphologically* or *phonetically* similar to those in the original query string. Other techniques are aimed at obtaining *semantically* relevant candidates to search for. These include finding:

- Synonyms: Terms obtained by looking the original word up in a thesaurus and taking the corresponding synonyms.

- Antonyms: Terms with opposite meanings.
- Superconcepts: Terms with more general meaning.
- Subconcepts: Terms representing special cases of the more general term.

Generating these semantically similar candidates requires dictionaries or knowledge bases. Important types of dictionaries include lexicons, thesauri, and domain-dependent knowledge bases.

When the above techniques fail to find a satisfactory number of relevant documents, some people have experimented with expanding the queries further using term re-weighting techniques such as relevance feedback [21].

3.2 INVESTIGATION IN A LARGE SOFTWARE SYSTEM

Before we apply normal query expansion techniques in large software systems, we should analyze the appropriateness of each of them, and determine if any other techniques are necessary to solve the special problems of searching in source code systems. In this section, we describe our investigation of some samples in a large software system. Then we derive our search strategies based on the analysis of the results of the investigation.

We selected 300 non-consecutive names from a database that contained hundreds of thousands of names used in a large telecommunication system. Then, we examined how each of the names was composed, and added it to one or more of the following categories according to what it contained. Finally we computed the percentage of each category in the 300 names as shown in Table 3.1.

This table is an indication of the most frequently used naming styles in source code systems. We put a name into a category of the styles in the table if it contained:

- **English words only**: This category is composed of names in which there were no abbreviations. Also, all of the words, which might or might not be stopwords, were clearly separated by word delimiters such as underscores, and were words in a typical English dictionary.

In our sample, 43% of the names contained only pure English words. That is, 57% of the names contained at least one abbreviation. This is probably one of the biggest differences between source code and ordinary documents.

Formats Contained in Names	Percentage
English Words only	43.00%
Stopwords	10.00%
Acronyms	46.67%
Truncated Words	48.33%
Words with Vowels Removed	10.33%
Affixes	18.00%
Phonetic Variations	10.00%
Other Abbreviations	22.25%
Contracted Abbreviations	30.67%

Table 3.1 Percentage of the names containing various word formats

- **Stopwords**: In this category, there were some common English words in the name, such as “out” in "check_out". Sometimes these stopwords played an important role in the names. Removing the stopwords, or replacing them with other stopwords might produce something very different. For example, in the "check_out" case, a name with only "check" might mean something far from "check_out", while "check_in" might show an opposite operation.

- **Acronyms**: Here, there were some abbreviations in the name that were the acronyms of some phrases. In this experiment, about 47% of the names contained acronyms, which meant that programmers of the system used a large number of acronyms in their source code.

- **Truncated words**: In this category, there were abbreviations in the name that were computed by taking the first few letters of a word and removing the rest. For example, people frequently used "init" instead of the whole words of "initial" or "initialize". Surprisingly, these truncated form of words occurred in about 48% of the names. Therefore, this is another important way to obtain abbreviations.

- **Words with vowels removed**: Some names obviously contained a word, from which all the vowels except the first letter were removed. Sometimes this all-consonant form of a word was also truncated so that only the first few letters of it appeared in the name. For example, "debug" could be abbreviated as "dbg", and "utility" could be abbreviated as "utl".

- **Affixes**: The English words in the names had prefixes and/or suffixes. For example, "ing" and "ed" were the most popular suffixes in the system. Prefixes, which only appeared in 1% of the names, were in fact a part of a few commonly seen words, such as "in" of "invalid".

- **Phonetic variations**: An abbreviation could be formed by picking a few consonants of a word in such a way that the abbreviation was still pronounced like the original word. For example, "message" was usually abbreviated as "msg" in our sample names.

- **Other Abbreviations**: Some abbreviations were also formed in other ways. For example, "tty" stood for "teletype", and "ksh" stood for "korn shell". It is very hard to describe a general model for how such names are derived. In fact, they are also among the abbreviations from which there is no way people can judge their meanings, if they do not already know them. Fortunately, these abbreviations, although occurring in more than 20% of the names, were usually the widely used terms of the domain.

- **Contracted Abbreviations**: About 30% of the names contained several abbreviations and/or words, between which there were no word delimiters. For example, "listdbg" was a combination of "list" and "dbg", in which case, it was very easy for human to infer that it must represent "list-debug". However, in many cases, it was very hard for both human and the computer to break the name correctly.

Besides the naming style, we also discovered that people tended to use the shorter version of words. Table 3.2 shows among the abbreviations we examined above, the percentage of abbreviations with different lengths. From this table, we can see that the vast majority of abbreviations were three or four letters long. Some abbreviations were two or five letters long. There were no abbreviations longer than six letters in our samples, but we did find several such abbreviations in the dictionary of the abbreviations of the system, as well as in the source code. Therefore, abbreviations were mostly two to five letters long, and those with six letters or more existed in the system, but were rarely seen.

Length of the abbreviation (number of letters)	**Percentage**
2	13.25%
3	56.3%
4	30%
5	5%
>=6	0%

Table 3.2 Percentage of abbreviations with different lengths

3.3 SELECTION OF INTELLIGENT SEARCH TECHNIQUES FOR LARGE SOFTWARE SYSTEMS

We can see from the above analysis that most of the methods of computing morphologically similar words in normal systems have been used in software systems. However, because of the presence of abbreviations all over the systems, we need to adapt some of the techniques to this problem. We might also need other approaches to compute or decompose abbreviations.

3.3.1 Regular text operations that are adaptable to source code systems

First of all, we can use word delimiters to split a word into multiple words. However, in source code systems, changes in the case (lower to upper) of letters, numbers and other special symbols can serve as word delimiters as well as the spaces and hyphens found in normal documents. For example, "CallForwardBusy" is a typical name in many source

code systems; we can separate it to "Call", "Forward" and "Busy" according to the change in case.

Secondly, similar to ordinary systems, software systems allow stopwords. Sometimes stopwords appear in the names as distinctive units, and some other times their initials are taken to compute the acronyms. Therefore we can not always ignore stopwords. For instance, if we ignore the stopwords in a query, we should consider them a factor to evaluate the results. And besides, when we do not know what is the acronym of a phrase, which is usually the case in automated systems, we should compute the acronyms in at least two ways, with or without stopwords. Sometimes only one of the acronyms will occur in source code, sometimes both.

As we have seen in the above study, acronyms and truncated words are the most popular abbreviations in source code systems. Furthermore, removing vowels and then truncating the results to shorter versions are also commonly used to derive abbreviations. Consequently, if we derive the acronyms of the phrases, and truncate the words and their all-consonant formats, we should be able to compute most of the abbreviations used in the system.

Of course, the length of the abbreviations is an issue. According to our experiment, abbreviations often contain two to five letters, and thus our solutions should cover abbreviations with such a length. For instance, when we truncate a word or its all-consonant format, we should at least take the first two to five letters to generate different abbreviations of the word. Longer abbreviations can be obtained by taking the first six letters or more. Considering that we will be looking for names with the computed abbreviations of the query as the names' substrings, shorter abbreviations generated by truncation in fact can retrieve all the names found by the longer ones. For example, "db" can certainly retrieve all the names having "dbg" as one of their substrings. Therefore, a six-letter long truncation has covered the abbreviations beyond six letters, and furthermore, an abbreviation with only the first two letters of a word can do all the job assigned to longer versions. It seems that truncated abbreviations that are two letters long have solved all the problems that can be solved by truncation, and consequently, we do not need other abbreviations to find additional results. However, an abbreviation with only two letters might retrieve a huge number of results, too, and thus lead to very poor precision. In order

for us to evaluate the effectiveness of the abbreviation generation methods and to select the most efficient ones to use, as will be discussed in later sections, we will use all the computed abbreviations in our query expansion before the experimentation and evaluation stage. As a result, we will generate the abbreviations with a length from two to six.

3.3.2 Regular text operations that are not used

Other than the above methods, we also realize that stemming and soundex are not necessary or not appropriate for source code systems.

Stemming is the method to remove affixes in normal query expansion systems. Although affixes are used less frequently than in normal documents, they are not rare in source code systems, either. We discover, however, that if we use the truncation methods discussed above, and the truncated words are short enough, they can already retrieve all the possible variation of a word with affixes. For example, given "terminate", its first six letters are sufficient for searching "terminator", "termination" or "terminating".

We have mentioned above that we will reduce the words to as short as two letters long by truncation, therefore, stemming will not be able to find many additional results after searching for truncated words. In fact, stemming will only make a difference on some irregular verbs. For example, "lying" is the present participle of the verb "lie". Stemming can correctly compute "lie" from "lying", while truncation can only get "ly" and "lyi". These cases may sparsely exist in source code systems, but we have not seen one in our experiment. Thus we expect the cost-effectiveness of this technique will be too low for source code systems, and do not use it as one of our query expansion methods. We will leave stemming in source code systems to future studies.

Soundex [24] was originally developed to deal with the difficulty of searching peoples' names, where users might not know the exact spelling. The Soundex algorithm computes a code for every word, and words with the same code are supposed to be spelled differently, but pronounced similarly. Therefore, Soundex can help us find spelling variants of a given name.

However, soundex is not very useful in source code systems because first of all, people usually know how to spell the words they are looking for, unless in some rare cases when

they have to guess and search for names that they have only heard from other people. Secondly, the soundex rules were not designed specifically for source code systems, and some of the rules have shown to be inappropriate for our purpose. As a result, although programmers have used some phonetic variation of the words in source code systems, the soundex rules may not help us compute the right variation. Therefore a soundex tool designed specifically for source code systems will be more useful for us, and we will leave it for future study. Finally, similar to the affixes, many phonetic variations of words in source code systems can be computed by truncation, or they are so commonly used that most programmers already know the particular variations, such as "msg" stands for "message", and "svr" stands for "server". So we do not adopt the soundex method, either.

3.3.3 Additional text operation techniques for software systems

It should be noted that although most of the normal query expansion techniques can be adapted to the special requirements of software systems, the 30% of names containing contracted abbreviations still remain unresolved. When people query these strings, and expect to find some related names, our intelligent search tool should correctly identify every word in the string before further computation and query.

However, the task of splitting contracted abbreviations is intrinsically hard. Lethbridge and Anquetil [3] have introduced a series of methods for generating all the potential abbreviations in a software system, and have used the generated abbreviations to decompose file names. This idea is useful for us, too. If we have a list of abbreviations used in a system, we will be able to correctly decompose most of the contracted abbreviations. For example, given a contracted abbreviation "listdbg", and we find that "dbg" is an abbreviation used in the system, we know that it probably should be split as "list-dbg". Of course, we might also find some other suitable abbreviations, such as "std", which breaks "listdbg" as "li-std-bg". We accept all these decompositions because altogether there are only a few possible alternatives, and we do not want to choose one of them arbitrarily.

The abbreviation generation methods shown in Table 2.1 also give us a hint of the decomposition. Many abbreviations are computed by removing the vowels except the first letter. Hence, the only chance that a vowel appears in the abbreviations formed this way is

when the vowel is the first letter of the word. For instance, "actual utility" can be abbreviated and contracted as "actutl". Thus given "actutl", and if we break it in front of the vowel "u", which will be "act-utl", we might get the correct decomposition.

Of course this vowel-heuristic does not work when no vowel is present, in which case, we believe simply breaking after every consonant should be able to find most of the possibly related names, along with a large amount of noise. For example, if "dbmgr" is broken as "d-b-m-g-r", we can find "database_manager", as well as "db_msg_handler". Therefore, if the first method fails to suggest the correct decomposition, the vowel-heuristics and this consonant-heuristics should be regarded as complements.

When all of these methods do not help at all, knowledge bases are probably the best choice, as described in the following section.

3.3.4 Other methods

Besides all the above methods that compute the morphological variations of the query, we have also mentioned the usage of the semantically related concepts in normal systems. Conceivably, synonyms, superconcepts and subconcepts are all suitable for source code systems as well. Looking up in the dictionaries that contain these semantically related terms is often fast with only a few but high quality returns, and is especially useful when the above methods can not offer us good solutions.

The concern is which dictionaries to use. We believe that domain-dependent knowledge bases are more effective than normal dictionaries or thesauri, whose vocabularies cover too many items that are not used in source code systems, yet lack of the domain knowledge and the special representations in a specific system. It is typically difficult, though, to find or build such a knowledge base. Fortunately, as we have mentioned in section 2.4.1, some of our previous studies [22] have built a knowledge base, from which we can derive the relationship between the concepts used in the system we are working with. It is small, specific, and easy to maintain, and therefore, we choose this knowledge base, rather than ordinary dictionaries, for our intelligent search tool.

It should be noted that we are not going to look for antonyms of a query in our intelligent search because we doubt their appropriateness in general for search purposes. We leave this to future studies.

3.4 SUMMARY OF INTELLIGENT SEARCH TECHNIQUES USED IN OUR STUDY

We have discussed our selection of intelligent search methods for large software systems, each of which might be used independently, or together with other methods to compute query variations. These methods can be put into the following three categories:

Computing abbreviations:

- Separating words according to capital letters.
- Removing numbers and special symbols.
- Removing common words.
- Generating acronyms.
- Truncation.
- Removing vowels.

Decomposing contracted abbreviations:

- Decomposing using a dictionary of abbreviations.
- Decomposing before vowels.
- Decomposing after every consonants.

Finding related concepts:

- Synonyms.
- Superconcepts.
- Subconcepts.

CHAPTER 4. CANDIDATE GENERATION ALGORITHMS

This chapter explains the candidate generation stage of the intelligent search tool as shown in Figure 2.3. At this stage, we use the text operation methods selected in Chapter 3 to compute the variants of the original query string, and generate additional search candidates based on the variants. Meanwhile, we also produce and collect a set of evaluation strings for later use. The explanation starts from the definition of some concepts, and then illustrates the candidate generation procedure step by step.

4.1 DEFINITIONS

From now on, the following concepts are frequently used in our description of the algorithms and experiments. Some of them refer to the same meanings as they do in normal documents, while some others are special denotations in our system.

Although all of the concepts below (except the ones for “entity name” and “query”) are mainly defined for describing the text operations performed on a query string, the same concepts, and hence the same text operations, apply to entity names as well. For example, we could extract keywords and stopwords from both a query and an entity name. We could also say that “the query is a multi-word term”, and “the entity name (or result) is a single-word term”.

4.1.1 Basic definitions

- **Entity name**: The name of an entity in a software system. Normally, there is no space in the name, but it could contain other word delimiters.
- **Query** or **query string**: The user input of a search. The query can be a phrase of natural language, such as “call forward while busy”, or a sequence of characters that are similar to an entity name, such as “CallForwardWhileBusy”. The tool treats the two formats in the same way, i.e. it retrieves the same results for both queries.
- **Word**: Any English word that can be found in an English dictionary. A word is the smallest meaningful unit of the entity names in source code, no matter in which format

it appears in the names, or whether there are word delimiters to separate it from other words. For instance, "CallForwardBusy" contains three words, and "listdbg" contains two (in the case where it is split as ‘list’ and ‘dbg’).

- **Abbreviation**: Any short format of a word or a phrase used in source code.
- **All-consonant format of a word**: One abbreviation of a word. It is computed by removing all vowels of a word except the first letter.
- **Stopword**: Common words as defined in normal information retrieval systems. Usually stopwords occur in source code as a whole. We do not consider the abbreviations of stopwords, unless they are parts of acronyms.
- **Keyword**: Any words other than stopwords. They can be abbreviated in source code.

4.1.2 Types of strings in source code

Essentially, the original query strings provided by the searchers, as well as the entity names in source code, can be cast into two categories:

- **Multi-word term**: Strings with spaces or other word delimiters that clearly indicate the separation of words. For example, "CallForwardBusy" is a multi-word term.
- **Single-word term**: Strings in which word delimiters do not indicate the separation of words. For example, both "_list" and "listdbg" are single-word terms.

4.1.3 Types of evaluation strings

Although evaluation strings are used in the evaluation stage, they are generated together with the search candidates. Thus, we also introduce them here:

- **List of related terms**: A set of strings that are relevant to the whole query string, e.g. synonyms, acronyms, superconcepts, subconcepts, etc. Although the acronyms might not be defined in the knowledge base that shows the conceptual relationship between strings, they represent the same meanings as the query string does. For example, “cfwb” refers to exactly the same thing as “CallForwardWhileBusy” in the system we are working with. Thus acronyms should also be in this category.

- **List of words**: The set of the words appearing in a query string or in an entity name. This list contains all the items in the list of keywords and the list of stopwords.
- **List of stopwords**: A collection of all stopwords in a string. They are collected mainly for evaluation purpose.
- **List of keywords**: A collection of all keywords in a string.
- **List of abbreviations**: A collection of computed abbreviations of a keyword.

The relationships among these concepts are shown in Figure 4.1; we will explain the numbers in more detail in the following sections.

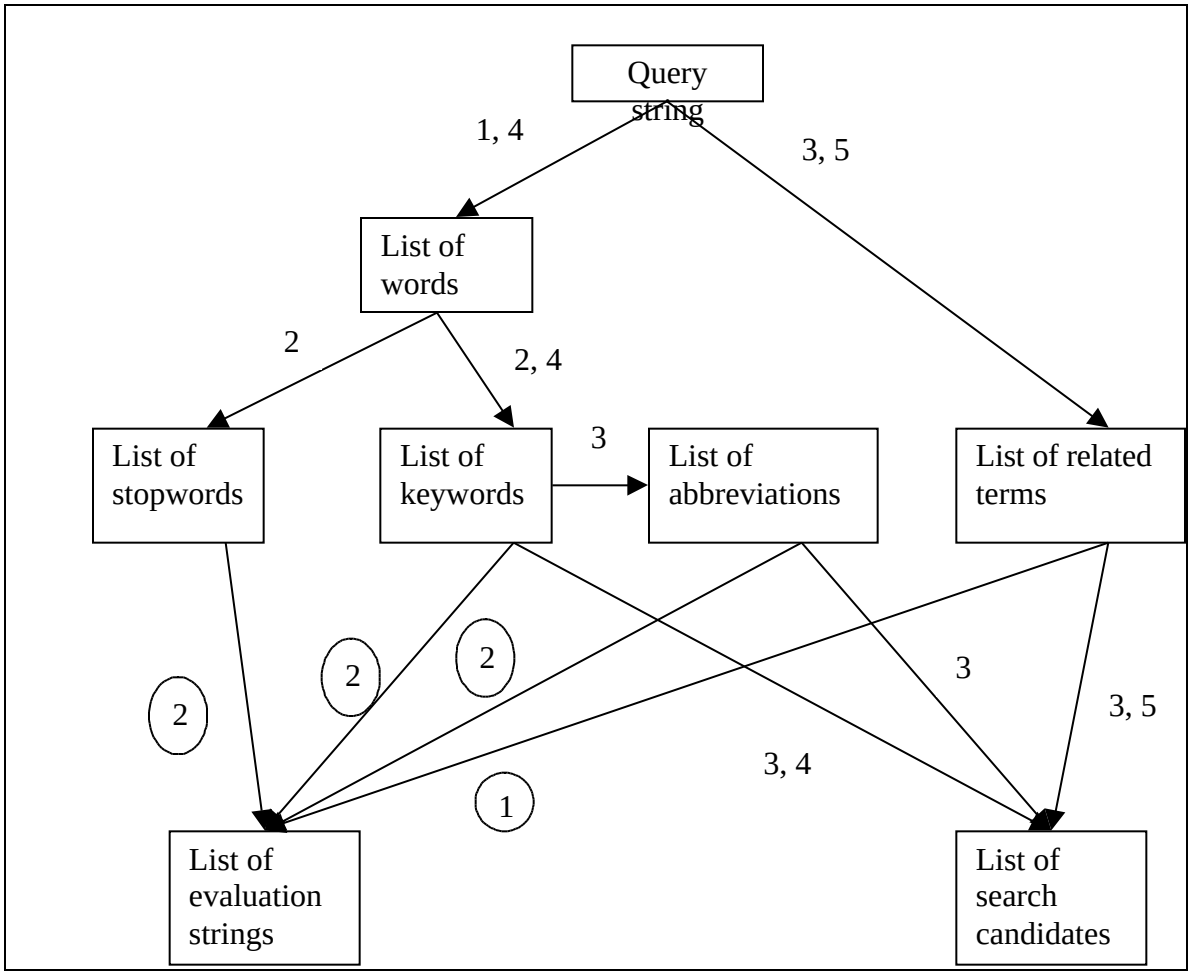

Figure 4.1 Relationship among the text operation objects.

4.2 General idea of the candidate generation algorithms

In addition to the relationship among the objects of the text operations, Figure 4.1 also demonstrates when and how the strings are formed. The numbers without oval shows in which steps of the candidate generation stage the corresponding transformations happen, whereas the numbers in an oval indicate which evaluation algorithms use the corresponding strings, which will be the focus of the next chapter.

During the process of candidate generation, we tend to create for each query its semantically relevant search candidates generated by computing the morphological variants or consulting the dictionaries. In the following sections, the explanation of the algorithms focuses on the morphological variants only, since the dictionary look-up techniques are relatively simple and are similar to those used elsewhere.

The general procedure of computing the morphological variants of the query string consists of the following steps: 1) identifying the keywords in the query, 2) computing new search candidates by concatenating the keywords together using different sets of abbreviations every time, 3) computing new search candidates by taking one of the keywords only. The overall approach therefore starts with the identification of word delimiters and the separation of the words in the original string.

The differences between the algorithms of computing morphological variants depend on the type of the given query strings. As defined above, there are two types of query strings, multi-word terms and single-word terms. Normal text operation methods work better on multi-word terms, while the specifically designed abbreviation expansion techniques are more useful for single-word terms. Therefore, different sequences of intelligent search approaches are applied to these two types of original queries.

The candidate generation stage contains five steps. The first two steps are for preparation, while the last three are the actual algorithms that generate search candidates: Step 3 is a sequence of abbreviation *concatenation* methods that are suitable for both multi-word terms and single-word terms; Step 4 includes abbreviation expansion methods, which are applied to single-word terms only; Step 5 involves the knowledge bases for searching the semantically related terms, and is useful for both types of query strings.

4.3 CANDIDATE GENERATION ALGORITHMS

4.3.1 Preparations

Step 1: Separation of words

- **Techniques used**: This step uses word delimiters to isolate individual words in the query, then changes the words to lower case.

- **Action**: The following are what we regard as word delimiters in source code systems:
  - *Common word delimiters* such as spaces, underscores, dash, period, and so on.
  - As discussed in Chapter 3, in case-sensitive environments an *upper case* letter embedded in a sequence of letters usually indicates the beginning of a new word within a string.
  - *Numbers* are also good places to split words. For example the variable name 'sys5response' would be split into 'sys' and 'response'. We can then search for 'sys*response'. This improves recall, but results in a potential loss of precision. For instance, 'sys5' might be something very specific and a result like 'sys4response' might not be highly related. However, considering that query expansion is meant to bring more results to users, we would like to keep these results and let the users decide whether they are useful or interesting.
  - *Other non-alphabetic characters*, though less seen than the above three cases, are treated as word delimiters if they appear in the query. Such symbols can be '@', '&', '$', and so on.

  After word separation, we also transform all upper case letters to lower case because our back-end database stores all names in lower case only.

- **Output**: The intermediate output of this step is a *list of words* extracted from the original query string. For later evaluation, we also store the original query string itself as the first member of the *list of related terms*.

- **Example**: If the original string is "CallForwardWhileBusy", we get "call", "forward", "while" and "busy" at this step, and put them into the *list of words*. Moreover, we add "CallForwardWhileBusy" to the *list of related terms*.

Step 2: Removal of stopwords

- **Techniques used**: Removing the stopwords by consulting the dictionary of stopwords.
- **Action**: We check every item on the *list of words* against the dictionary of stopwords, and split the *list of words* into two new lists, i.e. the *list of stopwords* used in the original query string, and the rest, the *list of keywords*. From now on, we will primarily work with the keywords until the time when we need to know the removed stopwords to help evaluate the relevancy of the query results.
- **Output**: The intermediate outputs of this step are the *list of stopwords* and the *list of keywords*.
- **Example**: In the above example, the *list of stopwords* contains "while", and the *list of keywords* contains "call", "forward" and "busy".

4.3.2 Abbreviation *concatenation* algorithms (ABBR)

Step 3: Concatenating abbreviations for the original query string

- **Techniques used**: This algorithm generates additional search candidates by computing acronyms or using the abbreviation concatenation methods.

- **Action**: The basic idea is to produce new search candidates by concatenating every keyword, which may be abbreviated differently in different candidates. The following are the sub-algorithms that compute new search candidates:

  Step 3.1: Computing the acronyms (ABBR_ACR) by picking the initials of each word in the *list of words*, or in the *list of keywords*. In the above example, "cfwb" and “cfb” are new candidates.

  Step 3.2: Concatenating keywords in a regular expression (ABBR_KEY) in the form of "keyword1*keyword2*keyword3". So "call*forward*busy" is a new

candidate. (We use simple regular expressions where * represents any sequence of characters).

Step 3.3: Concatenating abbreviations of keywords in a regular expression (ABBR_CUT). The algorithm first generates a list of abbreviations for each keyword. With these abbreviations, we can generate a group of candidates, each of which comprises one abbreviation format from every keyword, and is a regular expression in the form of "abbrKW1*abbrKW2*abbrKW3". The following sub-steps are the two ways we generate the abbreviations for each keyword:

Abbreviation generation algorithm:

· Step 3.3.1: Removing the vowels except the first letter.

For example, "call" becomes "cll" and "forward" becomes "frwrd". "cll" and "frwrd" are the all-consonant format of the corresponding keyword.

· Step 3.3.2: Truncating the keywords and their consonant formats. Each new abbreviation consists of the first few letters of the keyword or its consonant format. As discussed in last chapter, the abbreviations of the keywords are 2 to 6 letters long.

As a result, each keyword can have up to 11 abbreviations: one is its consonant format; half of the remaining ten are from truncating the keyword itself; and the other half are from truncating its consonant format.

For example, the abbreviations of "forward" can be "forwar", "forwa", "frwrd", "frwr" and so on.

Therefore, the intermediate output of step 3.4 is a *list of abbreviations* for each keyword.

For the ongoing example, overall we have "call*frwrd*busy", "cll*forw*bsy", "ca*fo*bu", and so on, as additional search candidates.

Step 3.4: Search every single keyword (ABBR_SNG). For all members of the *list of keywords*, we search for entities containing at least one of the keywords.

- **Output**: Step 3 generates a list of search candidates using acronyms and contracted keywords. It also generates a *list of abbreviations* for every keyword.
- **Example**: "CallForwardWhileBusy" can have new search candidates such as "cfb", "call*forward*busy", "call*frwrd*busy", "ca*fo*bu", “busy” etc. Each keyword is associated with its abbreviations, for instance, "forward" is linked to "frwrd", "frw", "forw", and so on.

4.3.3 Abbreviation expansion algorithms (EXPN)

Step 4: Expanding potential abbreviations

- **Techniques used**: This algorithm uses the abbreviation expansion techniques to decompose the single-word query string.

- **Action**: This step only applies to the single-word terms– i.e. query strings that can not be decomposed into multiple words using the above methods, and which hence have only one item on the *list of words* after the Preparation stages (Step 1 and Step 2). For example, if the original query string is "actutl" or "listdbg", the above methods can not split it into several components.

We assume that, in this case, the single-word term may be a combination of abbreviations of several different words, so we try to split it. Conversely to what we did in Step 3, we need to find out possible methods to *expand* the abbreviation or abbreviations.

The following are the approaches to the expansion, from which we form regular expressions as new search candidates:

Step 4.1: Searching for the regular expression that contains the whole query string (EXPN_WHL). For instance, in the "listdbg" example, we would search for "*listdbg*". This is like searching using grep.

Step 4.2: Breaking the abbreviation in front of vowels (EXPN_VOW) that follow a consonant. For example, if the original string is "actutl", one way to split it

is "act-utl". In the above example, "listdbg" is split as "l-istdbg". So "l*istdbg" is taken as a search candidate.

Step 4.3: Breaking the abbreviation after every consonant (EXPN_CON). For the ongoing example, we create another search candidate as "l*is*t*d*b*g*".

Step 4.4: Breaking the abbreviation using the dictionary of abbreviations (EXPN_ACR) from the knowledge base. By going through the dictionary of abbreviations used in this system we obtain a list of abbreviations, which appear in this query string. These abbreviations are then used to split the string and generate search candidates. In the above example, if the list of abbreviations contains "db" and "std", “listdbg” can be split as “list-db-g” and “li-std-bg”, hence the new search candidates will be "list*db*g" and "li*std*bg".

Keyword generation algorithm:

One problem that remains until now is that there is only one member in the *list of keywords* for this type of query string after Step 3. After the above operations in this step, we have somehow broken up this string, consequently each part of the decomposed string should be treated as a potential key unit of the original string. As a result, the *list of keywords* grows by adding all the unique parts from all candidates found in this step, except for the very short parts that have only one or two letters.

For instance, when searching for "listdbg", we would already have search candidates "l*istdbg", "l*is*t*d*b*g*", "list*db*g" and "li*std*bg" from the above operations of this step. Now we would add, "istdbg", "list", and "std" to the *list of keywords*; “listdbg” should also be on the list since it was added in Step 2. Each of these keywords, except for the original string itself which has been taken care of in the first three steps, generates additional search candidates in the following ways:

Step 4.5: Searching for each of the keywords (EXPN_KEY). In the example, "*istdbg*", "*list*" and "*std*" are new search candidates.

Step 4.6: Breaking the keyword after every consonant (EXPN_KEY_CON). Just as what we have done in EXPN_CON, from "std" we get "s*t*d*" in the same example.

- **Output**: This algorithm produces additional search candidates for single-word terms using abbreviation expansion methods. Various expanded versions of the query also help forming the list of keywords of the single-word term.
- **Example**: "listdbg" has extra search candidates such as "list*db*g", "li*std*bg", "dbg", "std", and etc. The *list of keywords* thus contains "list", "std", "dbg", and so on.

4.3.4 Algorithms using knowledge bases (KB)

Step 5: Looking for related concepts in the knowledge base-.

- **Techniques used**: This algorithm looks up the query string in the knowledge bases to find its semantically related concepts.
- **Action**: From the knowledge base we have on hand, we can obtain the synonyms, as well as superconcepts and subconcepts of the original query string. In addition, we look for concepts containing every item on the *list of keywords* as well for multi-word query strings, if the original query string is not found in the knowledge base. Finally, all these related concepts are added to the *list of search candidates*, as well as to the *list of related terms*.
- **Output:** The list of search candidates and the *list of related terms* have new members found in the knowledge base.
- **Example:** If "cfb" is a synonym of "CallForwardWhileBusy", "cfb" is considered a new search candidate and is added to the list of search candidates (if it has not been a member of the list). Moreover, "cfb" should be appended to the *list of related terms* of "CallForwardWhileBusy".

4.4 SAMPLE OUTPUT OF THE CANDIDATE GENERATION ALGORITHMS

Table 4.1 and Table 4.2 show examples of the candidate generation procedure. Table 4.1 lists the sample output of "CallForwardWhileBusy", a multi-word term, and Table 4.2

lists that of "listdbg", a single-word term. The major difference between the two samples is that Step 4 is not applicable for the multi-word term.

Step	Intermediate Output	Search Candidates Generated
1	list of words: call, forward, while, busy. list of related terms: CallForwardWhileBusy	
2	list of keywords: call, forward, busy. list of stopwords: while.	
3	list of related terms: cfb, cfwb list of abbreviations of each keyword: call→cll, cal, ca, cl forward→frwrd, forwar, frwr, forwa, … busy→bsy, bus, bu, bs	cfb, cfwb, call*forward*busy, cll*frwrd*bsy, cal*frwr*bu, ca*fo*bs, busy, …
4	(n/a)	
5	Concepts including the keywords: call forwarding-busy internal source, …	Call forwarding-busy internal source …

Table 4.1 Sample output of candidate generation stage for "CallForwardWhileBusy".

Step	Intermediate Output	Search Candidates Generated
1	list of words: listdbg list of related terms: listdbg	
2	list of keywords: listdbg list of related terms: listdbg	
3	list of abbreviations of each keyword: listdbg→listdb, listd, list, lstdbg, lstdb, lstd, …	listdb*, listd*, list*, lstdbg*, lstdb*, lstd*, …
4	list of keywords: listdbg, istdbg, std, list, … list of abbreviations of each keyword: istdbg→istdb, istd, …; list→list, lst, …; std→st; …	*listdbg*, li*stdbg, list*dbg, li*std*bg, list*, s*t*d*, …
5	(none)	

Table 4.2 Sample output of candidate generation stage for "listdbg".

4.5 SUMMARY OF THE CANDIDATE GENERATION ALGORITHMS

Table 4.3 sums up the techniques used and the outputs produced in the candidate generation stage. The search candidates generated are used respectively in the searching stage to retrieve entities in the database, while the intermediate outputs are adopted all together in the evaluation of the retrieved results.

Step	Techniques Used	Intermediate Output	Search Candidates Generated
1	Transforming upper case to lower case; Removing numbers and special symbols	List of words; List of related terms	
2	Removing common words	List of keywords; List of stopwords	
3 ABBR	Acronyms; Concatenating abbreviations	List of related terms; List of abbreviations of each keyword	Acronyms Concatenation of abbreviations
4 EXPN	Abbreviation expanded without using knowledge base; Abbreviation expanded using a dictionary of abbreviations	List of keywords	Expansion of abbreviations
5 KB	Synonyms; Superconcepts; Subconcepts	List of related terms	Related concepts

Table 4.3 Candidate generation techniques and results.

CHAPTER 5. RATING ALGORITHMS

After collecting all the search candidates produced by the algorithms described in Chapter 4, the intelligent search tool retrieves the new search candidates one by one through the query module of TkSee. When finished with the search, the tool should evaluate every retrieved result, and prioritize them when presenting them to the user.

This chapter first reviews the ordinary methods to evaluate the search results, as well as their inappropriateness to our problem. Then we introduce our algorithms designed specifically for source code systems with many abbreviations.

5.1 REVIEW OF RELATED RESEARCH

The major task of search result evaluation is to assess the similarity between the result and the original query string. The similarity measures usually calculate either the semantic proximity or the morphological proximity between the source and the target. The ranking algorithms of conventional information retrieval typically judge a retrieved document based on its semantic relationship with the query. Meanwhile, there are also some methods to calculate the morphological distance between strings. We will briefly review both methods in the next parts of this section.

5.1.1 Similarity between the query string and a retrieved document

Many normal information retrieval systems evaluate the retrieved documents based on term frequency measurements [5]. These measurements involve two types of frequency of the terms in the query:

- The frequency of the terms that appear in the retrieved document. A document with many occurrences of the query terms is considered more relevant to the query than those with few occurrences of the terms. So the internal occurrence of a term has positive effect on the weighting of the document.

  In addition to the occurrence, sometimes the font size and the position of the occurrence also affect the judgment. For instance, in some search engines like

"google" [8], if a query term is bigger in size than other text in a document, or if it appears in the titles or sub-titles, the document gains weight in its relationship with the query.

- The frequency of the terms that appear in the whole document set. The more documents that contain the same term, the more common the term is. Therefore, a common term appearing many times in a document does not necessarily mean that it is extremely relevant to the term. On the other hand, if a document contains a term that is rarely seen in other documents, probably this is a document in particular about the subject represented by the term, and hence it is highly related to the term. Consequently, the commonality of the term has a negative effect on the weighting of the documents.

Term frequency measurements are reasonable for normal documents, and sometimes work well. However, they are not appropriate for source code systems. As mentioned earlier, the search targets of this information retrieval application are entity names. Comparing with normal text, names are mini-documents. Programmers usually have to balance the contents and the lengths of the identifiers when they name the entities. Or in other words, they tend to put enough information in the names while keeping them from being too long, and that is probably one of the reasons why there are so many abbreviations in software systems. As a result, it is almost impossible for a term to appear twice or more in one name. Therefore, the term frequency measurements are not suitable for such mini-documents.

5.1.2 Similarity between strings

Comparing our intelligent search results with the original query is like comparing two strings, as opposed to comparing a string and a longer text document. The similarity between strings can be judged conceptually and morphologically.

Conceptual proximity between strings is typically derived from their semantic relationships in a semantic network, which is constructed from a source such as WordNet [17, 28] or Roget's thesaurus [18, 20].

Morphological proximity between a pair of strings does not involve any external facilities such as thesauri. The similarity is often defined by a distance function. According to Baeza-Yates [4], the two main distance functions are:

- Hamming distance: If the two strings are of the same length, their Hamming distance is defined as the number of symbols in the same position that are different (number of mismatches). For example, the Hamming distance between “text” and “that” is 2.
- Edit distance: The edit distance is defined as the minimal number of symbols that is necessary to insert, delete, or substitute to transform a string to another. For example, the edit distance between “text” and “tax” is 2.

Naturally, the higher the morphological distance between two strings, the lower their similarity. This technique is especially useful in spell_checking tools, which help people fix spellings errors by recommending suitable replacements.

Unfortunately, morphological distance measures are not suitable for calculating the alphabetical similarity between the original query string and a result derived from the above intelligent search algorithms. For example, given a query “attgrpmgr”, “abs_max_attmgr_grp_request” is a result found by a query generated in Step 4 of the above algorithm. The distance between the two strings judged by the above methods is very large, but according to human assessment, they are highly similar to each other.

The idea of conceptual distance is useful in some cases when the result is from a search candidate found from the knowledge bases (Step 5 “KB” in Chapter 4). As an additional example of the above query, “dvcmgr” is a superconcept of “attgrpmgr” in this system, and this relationship is defined in our knowledge base. Therefore, they are conceptually close, although they look very different. However, we can not obtain such kinds of relationship through normal thesauri, rather, a smaller but more specific knowledge base, like the one we have, is more useful.

Therefore, no matter whether we want to compare the semantic or the morphological distance, we need to develop a new set of algorithms for judging the search results of the intelligent search tool.

5.2 EVALUATION STRATEGIES

5.2.1 Rationale of using evaluation strings

Because of the difficulty of using normal evaluation methods to judge the search results in source code, it is essential to acquire much knowledge that we can apply in the evaluation. As we have learned before, both the query and the entity names can be a mixture of natural English words, acronyms, and various other styles of abbreviations. To deal with these combinations, we have to apply a certain amount of intelligence to break them up before we can compare them.

The intelligence we add to our rating algorithms is based on the knowledge we obtain during the generation procedure of the search candidates as we discussed in Chapter 4. We have done our best to break entity names into components using the specially designed text operation methods. More specifically, the knowledge that we have learned about a query string is embodied in the *intermediate outputs* of the candidate generation procedure as shown in Tables 4.1, 4.2, and 4.3. The intermediate outputs for a query are the *list of words*, *list of keywords*, *list of stopwords*, *list of abbreviations* for every keyword, and *list of related terms*; each item on these lists of strings resembles the original query string in one way or another.

For instance, “dvcmgr” in the above example is semantically similar to the query string “attgrpmgr” since it is its *superconcept*, while each of “att, “grp” and “mgr”, which may be obtained by *abbreviation expansion* in Step 4, could be a keyword of "attgrpmgr".

Hence when it is hard to rate a result term directly against the original query term, these strings can be used instead to judge the level of relevancy or similarity of the result. We therefore, call these intermediate outputs *evaluation strings.* The evaluation strings are weighted strings that are relevant to the original query term, and are used to evaluate the query results.

5.2.2 Types of evaluation strings

From the above steps, we acquire two types of evaluation strings:

1) One type represents some aspect of the *whole* query string, such as the strings in the *list of related terms*. For example, superconcepts and subconcepts stand for broader or narrower scope of meanings of the query, while the original query string itself, its acronyms and synonyms represent similar concepts but are alphabetically very different.
2) The other type represents a *part* of the query string, such as the strings in the *list of words* (including both keywords and stopwords). For example, according to the way we broke it, every item in the *list of words*, no matter whether it is a keyword or a stopword, is a meaningful component of the query. Therefore the appearance of any word of the list in the result indicates that the result contains at least one component of the query. Considering the fact that at times the keywords appear in their abbreviation formats, the *lists of abbreviations* are also useful in judging the existence of such components.

Due to the difference of the amount of the information they bear, every type is suitable for evaluating a specific group of results, and hence is used differently in the rating algorithms. Figure 4.1 shows, through the numbers in an oval, the rating algorithms in which the evaluation strings are involved.

5.2.3 General idea

When it is hard to judge the closeness of a result directly against the query, the evaluation strings will bridge the gap. That is, in order to compute the similarity between a result and the query, we only need to know the similarity between the result and the evaluation strings, as well as the similarity between the evaluation strings and the query.

Figure 5.1 shows the general idea of the result assessment using an evaluation string. If we know that an evaluation string bears a proportion of the information in the query, and that the result contains the evaluation string, we can decide that the result holds the particular portion of information.

Furthermore, after we apply the same technique to every evaluation string in the whole set, we should be able to determine overall what proportion of the information in the query is contained in the result under investigation.

Consequently, the rating algorithm has to judge a result against every evaluation string before it can draw a conclusion based on all evaluation strings. In the next section, we present our considerations for every aspect of the evaluation.

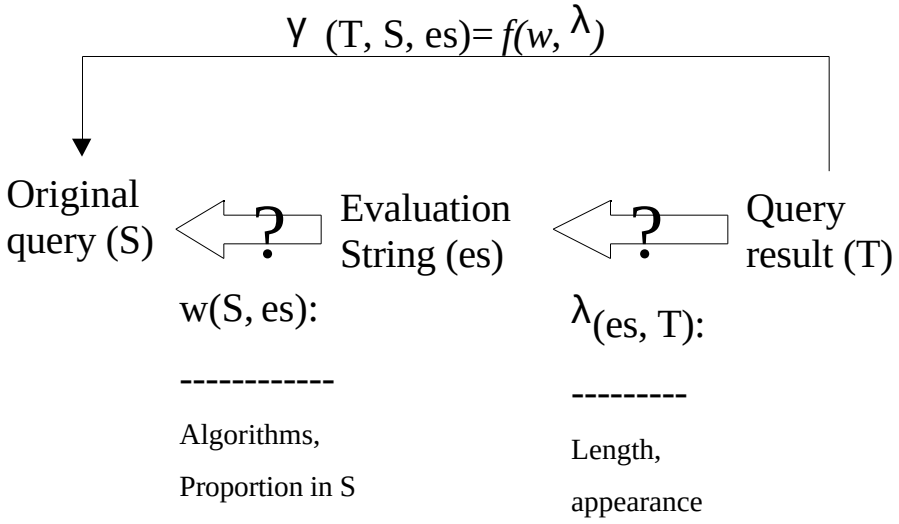

Figure 5.1 Using evaluation strings in the rating algorithm

5.2.4 Criteria for the evaluation

5.2.4.1 Judgment against one evaluation string

The evaluation strings carry the contents expected in the result under assessment. Nevertheless, we have to figure out how these evaluation strings can lead to a judgment. In general, the judgment depends on how good the evaluation string is, and whether it appears in the result. Lots of facts contribute to the answer to these questions. The following are what we believe to be the most important criteria, and what our rating algorithms compute:

1) Similarity between an evaluation string and the query: This can be defined as *the proportion of information contained in the evaluation string.* In order to show how good an evaluation string is, the major measure is how much of the contents of the query is borne in it. We consider that it is associated with the following two concerns:

- **Type of the evaluation string**: Every one of the first type of evaluation strings shares lots of contents with the *whole* query. No matter how they are alphabetically different from it, they conceptually have the same granularity of the query. Thus evaluation strings of this type intrinsically have a weighting of 1 in a scale of 0 to 1.

  Unlike those of the first type, each of the second type of evaluation strings conveys only a part of the information, because it is just one component of the query. In other words, these strings have smaller conceptual granularity than the query does. Thus evaluation strings of this type have a weighting of a fraction of 1; and this weighting depends on how many components the query has altogether.
- **Source of the evaluation string**: An additional element to differentiate one evaluation string from another is where they are from. Sometimes we have more confidence in certain good candidate generation algorithms, which also generate the evaluation strings, and hence more confidence in their outputs. For instance, a synonym probably is closer in meaning to the query than a superconcept, while a superconcept may be better than a short abbreviation computed by brutal truncation.

  Consequently, we need a way to represent our confidence in an evaluation string from a candidate generation algorithm. Yet confidence is hard to characterize using a fixed number; rather, we use two numbers to represent the range of the rating a result could get if it contains the evaluation string. The minimal value stands for the minimal rating that might be assigned to such a result, while the maximal value gives the upper bound.

In summary, the similarity between the evaluation string and the query depends on the type of the evaluation string, and the algorithms that produce it.

2) Similarity between the evaluation string and the result: This measure mainly judges the appearance of the evaluation string in the result, which is harder to estimate than one might first imagine.

First, we clarify what we mean by saying that one string appears in another. Typically, "A appears in B" means "A is a substring of B". According to this definition, A either appears in B or not. The decision can be represented by 1, which means "appear", or 0, otherwise.

However, being a substring is not a sufficient indication of appearance for our purposes. For instance, sometimes a word is *not* a substring of a result, but it does occur in it because the word is in one of its abbreviation formats. Many factors could affect the appearance measure. Thus, instead of giving an answer of exact "0" or "1", many times our algorithms produce a value between "0" and "1", and the following are the factors to consider:

- **The length of the evaluation string**: If a result contains either the evaluation string itself, or any of its abbreviations, the evaluation string is considered to *appear* in the result. However, the appearance of the evaluation string itself is believed to be more "complete" than that of its shorter versions, the abbreviations. Therefore, the longer a string occurs in a result, the more possible that the result contains the information the string carries.
- **Variants of the result**: The same length judgment also applies when the evaluation string does not occur in the result, but in the variant of the result. For example, if an evaluation string is "clstr", a result like "msg_cl_str" is considered to contain the evaluation string, because after removing the word delimiters, it is easier to see that "clstr" is really a component of the result. Take another example, suppose the query is "listdbg", and two of its evaluation strings are "list" and "dbg". Given a result like "list_debug", a normal substring measure only shows the appearance of "list". However, if we compute the all-consonant format of this result, which is "lst_dbg", we can easily see the existence of "dbg". Hence computing the variants of the result by applying simple text operations can also improve the accuracy of the judgment.

In summary, the similarity between the evaluation string and the result is a function of the length of its parts that appear in the result or the variant of the result.

3) The likelihood of the appearance: The presence of a word in a result is sometimes hard to define because the words are so short that they can be just a substring of other English words. For instance, “of” as a stopword itself, may occur in words like “office”. Moreover, “oos” as an acronym of “out of service” might be a part of “choose”, or be the abbreviation of other phrases. Hence, even when a word or one of its abbreviations is a substring of a result, it is possible that the substring does not represent the same meaning as the word does.

As a result, there needs to be a method that computes the *likelihood of the appearance* for an evaluation string to state its chance of occurring in the result, given that the evaluation string or one of its abbreviations is a substring of the result. The following two features of the specific substring are considered relevant to the *likelihood of the appearance* of the evaluation string:

- **Distinct independence of the substring**: One way to judge the likelihood is by checking whether there are any word delimiters in the result presented before and after the substring. If word delimiters help distinguish the substring from other parts, we are more confident that the word of the query represented by this substring appears in the result. Otherwise, the likelihood of this word occurring in the result is less than 100%.
- **Length of the substring**: Another way to judge the likelihood of appearance is through the length of the substring. Sometimes even if no word delimiters appear, a substring long enough can confirm the existence of the word or phrase it represents, because usually a longer string is less probable to be a part or an abbreviation of various words than a shorter one.

  For instance, it is relatively easy to decide that “forward” is a component of a result like “forwardmessage”. However, when it comes to a short word like "bus", things are different. It is hard to tell if the "bus" in a result like "servicebusy" is really the one we are looking for. Also, it is hard to judge which of the following results is

better in terms of “bus”, “msgbuserr” or “servicebusy”. Therefore, longer words and their longer abbreviations make us more confident than shorter ones.

So when we are evaluating the appearance of a short word or abbreviation, e.g. one with less than 4 letters, and if their appearances are not distinct (no word delimiters), we consider that the presence of this word in the result is a chance of less than 100%.

In summary, the likelihood of the appearance of the evaluation string in a result relies on the length and the independence of the part that is a substring of the result.

5.2.4.2 Judgment against the whole set of evaluation strings

Once the algorithm gives a rating against every evaluation string, the final rating of a result is based on the whole set of individual ratings. The following are the considerations that lead to the strategies of using the set of ratings:

- **List of the related terms**: If any item in the *list of related terms* appears in the result, probably the query string as a whole is a vital component of the result, because usually the entity names are so short that only important contents show up. Hence, the presence of any of the related terms indicates the strong relevancy between the result and the query. If the result happens to contain multiple related terms, which is rarely seen, the algorithms will choose the one that brings the highest rating. Therefore, among all the ratings from the first type of evaluation strings, the maximal value of them will be taken as the final rating for the result. And the whole evaluation process of a result will stop once a rating is obtained here.

- **List of words**: When no related terms are found, it is natural to assess the result by comparing the number of components it contains with that of the query. So, our algorithms should compute the appearance of all keywords and stopwords in order to show how many components of the query are present in the result.

- **Number of keywords present**: A result that contains more keywords is considered closer to the query in meaning than one that contains fewer.
- **Number of stopwords present**: Stopwords play an important role in source code systems, as well as they do in normal documents. If a query includes stopwords, the evaluation should reflect their presence, even though the candidate generation algorithms drop them to expand the query. For example, a query “out of service” may find “device_out_of_service” and “device_in_service”, in which case only stopwords can tell the difference between the results.

The sum of the appearance of all the components in the result embodies how much content of the query has been found. Hence, *addition* is the operation that is applied to the whole set of ratings given by the second type of evaluation strings.

Besides the above factors, there might be many other attributes that can differentiate one result from another; however, we do not want to consider all of them in our algorithms so that they turn out to be too complicated to follow. We believe the above thinking includes the most important aspects. If our later experiments show that the accuracy of the algorithms is not satisfactory, we shall adjust current attributes, or take more aspects into account.

5.3 THE RATING ALGORITHM

5.3.1 Denotations

In the rating algorithm, we use a set of symbols to denote the concepts or functions discussed above. First is the input of the rating algorithms:

S: the original query string;

T: the result under evaluation;

T': T's simple variations, such as with word delimiters removed, its acronym, or its all-consonant format;

ES: the set of evaluation strings, could be ES1 or ES2;

ES1: the set of evaluation strings of the first type, i.e. the list of related terms;

ES2: the set of evaluation strings of the second type, i.e. the list of words;

es: an evaluation string in ES;

A_{ij}: the *j*th abbreviation of the *i*th item of ES2;

W_{ij}: the similarity of A_{ij} to $ES2_i$;

L_X: the string length of a string X;

n: number of items in ES2;

m_i: number of items in the *list of abbreviations* of an *es* as the *i*th item of ES2;

The following are the values computed by the rating algorithms:

min_{es}: the minimal possible level of similarity between T and S based on es;

max_{es}: the maximal possible level of similarity between T and S based on es;

w(S,es): the similarity between S and es;

λ(es,T): the similarity between es and T;

ϕ(es,T): the likelihood of appearance of es in T;

γ(T, S, es): the similarity between T and S according to es;

R(T, S): the overall rating of the similarity between T and S.

These denotations represent the general symbols in the algorithms. As discussed below, the algorithms, based on different types of evaluation strings, share the same framework, but differ in details. Therefore, there are two versions of the above functions: one version for rating against ES1, the other for ES2. Correspondingly, the names of different versions of the functions are numbered as R1, λ1, w1, and R2, λ2, w2.

5.3.2 Evaluating a result in up to three steps

The evaluation process has three steps. The first two steps apply the rating algorithms utilizing the two types of evaluation strings respectively, while the last step handles the rest of the results that can not be evaluated by the evaluation strings.

The rating process of each result stops whenever it obtains a value other than zero in a step. As a consequence, it may need up to all three steps to complete the evaluation of a search result. :

$$R(T,S) = \begin{cases} R1(T,S,ES1), & if \quad R1(T,S,ES1) > 0 \\ R2(T,S,ES2), & if \quad R2(T,S,ES2) > 0 \quad and \quad R1(T,S,ES1) = 0 \\ R3(T,S), otherwise \end{cases}$$

(Equation 0-0)

Where R1, R2, and R3 each represent the respective algorithms used in the three steps.

5.3.3 A general framework for rating against evaluation strings

As a summary of Section 5.2 and an introduction to the next part of this section, the following formulas and pseudo-code show the general framework of the first two algorithms:

1) The **overall rating** is given against the whole set of evaluation strings:

$$R(T, S) = F(\gamma(T, S, es)), \text{ for all } es \in ES$$

(Equation 0-1)

The function first computes the rating of T given by every es, which is the γ function defined in 2). Then, the type of ES determines which function to apply on the whole ES set, as defined in Equation 1-1 and Equation 2-1.

2) **The similarity between T and S according to an *es***:

$$\gamma(T,S,es) = [min_{es} + (max_{es} - min_{es}) * w(S,es) * \lambda(es,T)] * \phi(es,T)$$

(Equation 0-2)

W, λ, and ϕ should all compute a value from 0 to 1. Hence, the γ function provides a value between min and max, depending only on the similarity between es and S (the *w* function), the similarity between es and T (the λ function), and the likelihood of es appearing in T (the ϕ function).

The *min*, *max*, *w*, λ values are very much dependent on the type of es. They will be discussed in detail later.

The ϕ function, the likelihood of the appearance of es in T is defined in 3).

3) **The likelihood of the appearance of es in T**, ϕ(es,T), can be described as the following pseudocode:

```
input: es, T;
t: T, or one of its variants T';

if es is not a substring of t
    return 0;
if es is a distinct individual of t
    return 1;
else
    if the length of es >= 4
            return 0.75;
    else
            return 0.5;
```

In this pseudocode, T' is calculated in 4). The judgment "es is a distinct individual of t" is given in the pseudocode as 3.1).

When es is a distinct individual of T or T', there is no doubt that it does not appear as a part of other word. Thus the likelihood of the appearance should be 1. Otherwise, for the es that we can not separate from other parts of T, we simply consider its prospect of appearing in T as either moderate or high, which has a value of 0.5 or 0.75 respectively.

3.1) **The distinct independence of es in T** can be simply judged by the following pseudocode:

```
input es, T;
if there are word delimiters before and after the substring es in T
    return true;
else
    return false;
```

The way to determine the word delimiters is the same as that in Step 1 of Chapter 4. A value of "true" indicates that es is clearly separated from other parts of T.

4) T's, **the variants of T**, are the set of strings obtained by applying any one of the following text operation methods on T:

- Removing word delimiters;
- Taking the all-consonant format of T;
- Computing the acronyms.

These are computed in the same way as the Step 3.1, 3.2 and 3.3 of the candidate generation algorithms in Chapter 4.

Besides the above conformity, the differences between Algorithm 1 and Algorithm 2 reside in the way we obtain *R, w,* λ, $\min_{es}$ and $\max_{es}$, in 1) and 2) of the framework. We will discuss these differences in detail below, plus Algorithm 3, the rating algorithm that does not use the evaluation strings.

5.3.4 Algorithm 1 (R1)

This algorithm examines the similarity between the result and the original query string using ES1, i.e. the *list of related terms*. Its purpose is to identify some of the best results quickly so that it does not have to analyze the components of S and T, as Algorithm 2 does.

1) **Overall rating of the result**. If we let R1 denote the rating of a result T against ES1, and suppose there are *n* strings on ES1, then R1, and hence R, is calculated by Equation 1-1:

$$R(T, S)=R1(T, S, ES1)=max(\gamma(T, S, es)), \text{ for all } es \in ES1$$

(Equation 1-1)

R1 takes the maximum value of the similarity between T and S against every item on ES1 as the overall rating of T against the whole ES1 set.

R1 will be 0 (zero), if none of the evaluation strings appears in T or T'. Or it will take the highest rating if one or more evaluation strings are contained in T or T', in which case, R1 is our final rating, and we will not proceed to Algorithm 2 or Algorithm 3.

2) In the γ function, which computes **the similarity between T and S according to an es in ES1**, the *w* and λ, min_{es} and max_{es} are defined as follows for any es in ES1:

- 2.1) ***min* and *max* values**: These values represent the extent to which an evaluation string resembles S. The weights should be from 0 to 1, where 0 stands for no resemblance at all and 1 means exactly the same. So we give a minimum and maximum value, min_{es} and max_{es}, to each of them to denote the range of its similarity. This weight depends on the source of the evaluation string, which is the candidate generation step that produces it. Of course, S is considered the best representative of itself, and hence it has a weight from 0.9 to 1; therefore, any results containing S will have a rating better than 0.9, and a result that is exactly the same as S will get a rating of 1.0. The rest have lighter weights. Initially, we set the value for each type of evaluation strings in Table 5.1. If later experiments show that the rating algorithms' performance is bad, we will adjust these values to improve the performance.

Source	min	max
Original query string (S)	0.9	1.0
Acronym (ABBR_ACR)	0.8	0.8
Synonym (KB)	0.75	0.75
Super concept (KB)	0.65	0.65
Sub-concept (KB)	0.65	0.65
Concept containing all keywords (KB)	0.65	0.65

Table 5.1 *min, max* value of items in ES1.

- 2.2) ***w*(S, es)**: The *w* function for items in ES1 is set to 1 because each one resembles the whole string S. So:

$$w(S, es) = 1, es \quad ES1$$

(Equation 1-2)

- 2.3) **λ(es, T):** We then check if T or T' contains at least one item from ES1 as its substring. If so, we consider T similar to one of the related terms of S, and thus similar to S itself. Otherwise, the similarity between T and the related terms must be zero.

Let λ*1*(x, T) denote the similarity between T and a string x in this algorithm. Then the similarity between T and an item *es* of ES1 is given by Equation 1-3:

$$\lambda 1(es, T) = \begin{cases} L_{es} / L_T, \text{ if es is a substring of T or T', and es} \ddot{e} \text{ ES1} \\ 0, \text{otherwise.} \end{cases}$$

(Equation 1-3)

5.3.5 Algorithm 2 (R2)

This algorithm examines the similarity between T and S using the *list of words* that contains the keywords, which are associated with their *lists of abbreviations*, as well as the stopwords. This makes sense. Since all the keywords and stopwords are extracted from S, each of them should represent a unique part of S. Thus, we believe that a result T

containing some of these parts must be partially similar to S, which contains all of them. Our second algorithm, therefore, is to calculate how many of the parts appear in a result.

1) **Overall rating of the result**. If we let R2 denote the rating of a result T given by this algorithm:

$$R(T,S) = R2(T,S,ES2) = \prod_{es \in ES2} (\gamma(T,S,es))$$

(Equation 2-1)

So R2 sums up for T the level of appearance of all the unique parts of S, and should provide a rating from 0 to 1. Again, if R2>0, it is the final rating R for T, and the evaluation procedure will not proceed to the next step.

2) In the γ function, the *w* and λ, min_{es} and max_{es} are defined as follows for an es in ES2:

- 2.1) ***min* and *max* values**: Given only a part of S appearing in T, the range of the similarity between T and S is very hard to determine. So we simply set a wide range, and will let later experiments tell us whether the range is appropriate. Initially, for every *es* on ES2:

$$min_{es} = 0.20,\ es \quad ES2;$$
$$max_{es} = 0.95,\ es \quad ES2;$$

- 2.2) ***w*(S, es):** Like in Algorithm 1, each evaluation string has its own weight in a range from 0 to 1. There should be a range of possible similarity between T and S if T contains an evaluation string.

  Unlike in Step 1, however, the sum of these weights should be 1 because each string stands for one part of S and they altogether form the whole string. Initially, we treat all these evaluation strings equally, no matter whether they are keywords or stopwords. Suppose there are *n* items in ES2, thus:

$$w(S, es) = 1/n, es \quad ES2$$

(Equation 2-2)

· 2.3) **λ(es, T):** We then check, for every evaluation string in ES2, whether it appears in the result T. Again, the appearance of es in T as its substring determines the level of similarity between them.

This function is defined differently for stopwords and keywords, because we do not abbreviate stopwords. Hence, the **appearance of a stopword** is solely determined by whether it is a substring of T or T':

$$\lambda 2(es, T) = \begin{cases} 1, \text{es is a substring of T or T' and es} \ddot{e} \text{ ES2;} \\ 0, \text{otherwise.} \end{cases}$$

(Equation 2-3)

Unlike a stopword, a keyword should also be considered as appearing in T if T or T' does not include the whole keyword but one of its abbreviations. Therefore, the **appearance of a keyword** is,

$$\lambda 2(es, T) = \begin{cases} \max(W_{ij}), \max(W_{ij}) > 0, \text{ for } j = 1 \text{ to } m_i \\ \text{if es} = ES2_i \text{ and } A_{ij} \text{ is a substring of T;} \\ 0, \text{otherwise.} \end{cases}$$

(Equation 2-4)

An evaluation string *es* appears in T if *es* is a keyword and one of its abbreviations is a sub-string of T or T'. Therefore, in Equation 2-4, $\lambda 2$ is mostly determined by another type of weight i.e. W_{ij}, the level of similarity between a keyword and its abbreviation. Such a similarity should be determined because an abbreviation can not always fully represent the whole word. We believe W_{ij} is

strongly related to L_{ij}, the length of A_{ij}, because the longer the abbreviation, the more possible it represents to the whole word. Initially we set:

$$W_{ij} = \begin{cases} 0, & L_{ij} < 2; \\ 0.3, & L_{ij} = 2; \\ 0.7, & L_{ij} = 3; \\ 0.9, & L_{ij} = 4; \\ 1, & L_{ij} >= 5; \end{cases}$$

(Equation 2-5)

Therefore, λ2(es, T) returns the weight of the longest abbreviation of an es of ES2 that appears in T. Again, we will let later experiments to show whether the settings are appropriate.

5.3.6 Algorithm 3 (R3)

At this point, most of the good results have been rated in the above two steps. The remaining results must be either very hard to evaluate or not relevant at all, because after the above steps none of them contains a meaningful component of S indicated by the evaluation strings, not even a sub-string with a length of 2 letters has been found that is an abbreviation format of S or its key components.

Conceivably, these results are found by querying "a*b*c*"-like candidates generated by the query expansion algorithms (EXPN) for the single-word query string cases. It is almost infeasible to judge the similarity between S and such kinds of result T. For instance, it is hard to come up with a good way to judge if a T "check_for_call_annc_button" is similar to the S "cfb". Hence we simply consider these results are not relevant at all. So all these results have a same rating, which is the lowest rating in our system, as expressed in Equation 3-1:

$$R3(T, S)=0.20$$

(Equation 3-1)

Result	Search Candidate	Algorithm	Rating by the algorithm	Rating by users
Busy_cfb_set	cfb*	R1	80%	85%
Dpn_nsi_call_forward_busy	call*forward*busy	R2	72.5%	60%
Serial_call_from_busy	Call*for*busy	R2	59.375%	35%
Agm_case_force_man_busy	ca*for*busy	R2	55.625%	25%

Table 5.2 Ratings of results for "CallForwardWhileBusy".

Result	Search Candidate	Algorithm	Rating by the algorithm	Rating by users
Listdbg_summ	List*dbg, li*stdbg,…	R1	95.83%	85%
Listmgr_write_debug	List*,l*is*t*d*b*g	R2	56.375%	65%
List	List*	R2	41%	50%
Cpacd2rte_enab_disab_rte_dbg	Dbg*	R2	40%	35%
Ls_reset	Ls*	R2	22.25%	30%
ac15_outpulse_delay_timer_t_type	Ls*	R3	20%	20%

Table 5.3 Ratings of results for "listdbg".

5.4 EXAMPLES OF RATINGS OF RESULTS

Table 5.2 and 5.3 show some ratings of the two query examples in Chapter 4. We will discuss these ratings in detail in the next chapter.

5.5 SUMMARY

Table 5.4 sums up the algorithms by comparing the values, methods or concerns for calculating each attribute. The shared cells between algorithms show their commonalities. Meanwhile, the numbers in brackets indicate the formulas that define the attributes.

Level in the framework	Attribute	Definition	R1	R2	R3
1)	**R(T, S)**	the overall rating	Maximum of γ (1-1)	sum of γ (2-1)	0.20 (3-1)
2)	**γ(T, S, es)**	rating against an es	min, max, w, λ, ϕ (0-2)		
2.1)	**Min**	minimal possible rating	Candidate generation algorithm	0.20	
2.1)	**max**	maximal possible rating	Candidate generation algorithm	0.95	
2.2)	**w(S, es)**	similarity between S and es	1 (1-2)	1/n (2-2)	
2.3)	**λ(es, T)**	similarity between T and es	L_{es} (1-3)	L_{es} , W_{ij} (2-3,4,5)	
3)	**ϕ(es, T)**	likelihood of the appearance of es in T	L, the distinctiveness of *es*		
4)	**T'**	Variants of T	Text operations		

Table 5.4 Summary of the rating algorithms.

CHAPTER 6. EXPERIMENTS

We performed a series of experiments to examine the performance of our algorithms. We first performed intelligent search against a selected group of query terms, comparing the quantity and the quality of the results obtained from the above candidate generation algorithms. After that, we evaluated the accuracy of our rating algorithms according to human judgment.

6.1 REVIEW OF MEASUREMENTS OF RETRIEVAL EFFECTIVENESS

The performance of an information retrieval system is typically measured by retrieval effectiveness, which is highly related to users' satisfaction with system output. There are many measures to retrieval effectiveness, among which *relevance* is the most frequently used one [11]. In inspection of the output of a query, an item's *relevance* to the user is usually indicated by his/her acceptance or rejection of the item, while the number of relevant items is used to calculate the effectiveness of retrieval.

6.1.1 Precision and recall

The most widely used measures of effectiveness that are based on relevance are precision and recall [4, 11], and they are defined as:

- **Precision** is the proportion of retrieved items that are relevant.

$$\text{Precision} = \frac{number\ \ of\ \ relevant\ \ items\ \ retrieved\ \ by\ \ the\ \ query}{total\ \ number\ \ of\ \ items\ \ retrieved\ \ by\ \ the\ \ query}$$

- **Recall** is the proportion of relevant items that are retrieved.

$$\text{Recall} = \frac{number\ \ of\ \ relevant\ \ items\ \ retrieved\ \ by\ \ the\ \ query}{total\ \ number\ \ of\ \ relevant\ \ items}$$

Both precision and recall range in value from 0 to 1. In general, precision goes up while recall goes down if the number of items retrieved by the query is reduced, and vice versa.

6.1.2 Drawbacks of precision and recall

In spite of their popular use, there are some major objections to precision and recall [11]. The first one is the difficulty of obtaining the total number of relevant items. When the query is based on a large collection of documents, this number is normally estimated by statistical techniques. Therefore, in these situations, recall is inexact.

Moreover, precision and recall don't take the difference among users into account. They assume that people have the same opinion on the relevance of a result. However, in the real world a retrieved item may be relevant to one user's query while being considered irrelevant to that of another, since everybody has different knowledge and information needs.

Furthermore, research has shown that "precision is not significantly correlated with the user's judgment of success" [29]. Because of these problems, we believe some user-oriented measures are more appropriate for us to evaluate the effectiveness of the tool. We might still mention precision and recall for general discussion purpose, but would not use them as a measure of our tool.

6.1.3 User-oriented measures

Korfhage [11] summed up the user-oriented measures that had been proposed. Among those measures, coverage ratio and novelty ratio are the most typical ones:

> *Coverage ratio: the proportion of the relevant documents known to the user that are actually retrieved.*
> *Novelty ratio: the proportion of the relevant retrieved documents that were previously unknown to the user.*

A high coverage ratio means the search has found most of the relevant items known to a user. A high novelty ratio means the search can reveal lots of relevant items that are new to

the user. Searches with high coverage ratio and high novelty ratio will increase the user's satisfaction with the search tool. Therefore in our experiments, we are going to use these two ratios as a part of the assessment of the performance of our tool.

6.2 METHODOLOGIES

We performed a series of experiments to evaluate the effectiveness of the tool, the performance of the search candidate generation algorithms, and the accuracy of the rating algorithms. Some of the experiments were conducted in our lab by running the tool in different situations without involving real users, while some other experiments used the actual software engineers to evaluate the query results of the tool.

6.2.1 Automatic evaluations

First of all, we obtained a group of searches actually performed by the software engineers maintaining a large telecommunication system. The searches were logged by users of TkSee during maintenance activities which took place during April, 2000.

We extracted from the log all the searches that produced only a small number of results. We worked with such searches since the fewer the results found, the more likely a software engineer would want to use intelligent search, in order to find additional results. Altogether we obtained 34 query strings with which to experiment; 14 of these contained multi-word terms, and the rest were single-word terms.

Then we performed a query for each string using the intelligent search process in a system that has more than 300,000 identifiers. The performance of each candidate generation algorithm was evaluated by comparing the average number of candidates generated and the portion of results found by each of them. Our rationale for this is that a good candidate algorithm should produce a number of useful results.

In addition, we calculated the average rating of the results generated from each candidate generation algorithm, where the ratings were given by our own rating algorithms. At this stage, the assessment was based on the whole result set of each query. None of the query returns was pruned.

6.2.2 Human evaluations

6.2.2.1 Procedure of the experiment

Next, we assessed the performance of our algorithms using *human* evaluators. We obtained the assistance of four volunteer software engineers in the system's maintenance group. Two of these participants had been working on the system for more than ten years. Another one had more than ten years of experience of software development, too, but had only been working in this system for about one year. The remaining one had been with this system for several months, and this was his first software development job.

We selected the 10 most typical query terms from the above 34 strings, and then randomly picked about 15 to 20 result terms for each of them. Altogether there were 178 result terms chosen for the 10 queries. Therefore, the questionnaire included the 10 selected queries, each alone with the results listed below it, as shown in Appendix A. The results were not listed in any specific order, and their ratings given by the rating algorithms were not shown to the participants.

For each query, the participants were first asked to rate their familiarity with it in a range from 1 to 5, where 1 meant no idea at all and 5 meant extremely familiar with it (Appendix B).

Then, for each result of the query, they were asked to rate the relevance, interestingness and their level of familiarity with it by answering "how useful is the result", "how interesting is the result" and "how familiar are you with the result" respectively. Again, the ratings were from 1 to 5, where 5 meant the highest level of relevance, interestingness or familiarity.

After rating each result, the participants were also asked to list other relevant strings that they knew, but were not found by the query. These missing results would be used to evaluate the general effectiveness of the tool.

The experiment lasted about 1 hour and 15 minutes. We first briefly introduced the research of intelligent search in large software systems, and the purpose of the experiment. Then the participants all read and signed the consent forms. Before we started the tasks, we also asked them to read the instructions carefully, and we explained every task in detail. Next the participants spent about 1 hour on filling out the questionnaires. All four

participants finished and handed in their questionnaires. After the experiment, we also had about 15 minutes of debriefing with every participant.

6.2.2.2 Usage of the experiment data

From the set of ratings of relevance, we calculated for each one of the 178 result terms its average human rating. Furthermore, according to the average human ratings, we grouped them into 5 categories, each represented one human rating level. We put the results with human ratings between 1 and 1.5 (including 1.5) in Category 1, which contained the least relevant terms. And similarly, results with average human ratings between 1.5 (not including 1.5) and 2.5 (including 2.5) appeared in Category 2, and so on. Category 5 consisted of the best results, rated more than 4.5 by the participants.

We used the human evaluation results to assess:

- **The performance of the candidate generation algorithms**. The number of results obtained by the algorithms in the above five categories helped us evaluate our analysis in the automatic experiments.
- **The performance of the rating algorithms**. First, we examined a few cases to see whether the rating algorithms computed reasonable grades for different results comparing with human ratings. Then, we analyzed the accuracy of the rating algorithms by comparing the ratings given by them with the average human ratings of the 178 results. Since ratings from the algorithms were from 20 to 100, and human ratings were from 1 to 5, we computed the rating accuracy for each result after dividing the rating from the algorithms by 20 times its average human rating.
- **The consistency of the rating algorithms.** We used the correlation coefficient between the ratings given by participants and by the rating algorithms, as well as the correlation coefficients between the ratings of different participants, to determine how the rating algorithms were consistent with human judgment.

6.3 OVERALL PERFORMANCE

6.3.1 Response time

The average response time of the 34 samples queries was 37 seconds, which is relatively long comparing with general user preferences of response time, i.e. 2 seconds. In multi-word cases, it was about 27 seconds, while in single-word cases it was 44 seconds. After further investigation, we learned that there were three major reasons that contributed to the long response time:

1) The long response time of the intelligent search tool was due to the infrastructure it is based on. The kernel of the infrastructure was a DBMS-like query mechanism, which was very efficient. Its optimized searching algorithms allowed instant response to the queries of strings starting with a non-wildcard character (e.g. dbg*). However, it was not very efficient when looking for patterns like "*d*b*g*" or "*dbg*", because there was no index for fragmentary strings and therefore the query mechanism had to scan the whole database. On average, it took about 1 second for the query mechanism to scan the whole database of hundreds of thousands of entries to retrieve such a regular expression.

Unfortunately, there were many search candidates in the pattern like "*d*b*g*" or "*dbg*", especially in the single-word cases. As shown in Table 6.1, on average the intelligent search tool generated for each query more than 30 search candidates, which were all such types of regular expressions that the query mechanism could not handle efficiently. Therefore, it needed more than 30 seconds for each query to scan the database for all the regular expressions.

Normal information retrieval tools such as the search engines of the Web have better performance because they use indexes like inverted files. As discussed in Chapter 2, these indexes make search very efficient. However, because the construction and maintenance of the indexes of the components of entity names, which were in fact fragmentary strings that could be as short as one letter in software systems, was very expensive, we did not adopt the "inverted files"-like index in our system. Nevertheless, even though the tool is less efficient than other tools, after all it is still much faster in searching a number of expanded queries than doing that using a series of "grep" commands.

2) In addition, the experiments we performed so far examined the performance of all the algorithms. All search candidates were treated equally, and no results were pruned. Thus, after the examination, we would be able to tell which algorithms outperform others because of the quality of their search candidates and results.

3) Also, in the 34 experiments we were retrieving all types of objects in the software system, such as files, routines, variables, constants, types, strings, and etc. In real world, people often only retrieve a certain type of objects in a query, e.g. files, routines, or variables, and each type accounted only a subset of the whole database. We found that the size of the subset affected the performance. For example, if the queries were looking for files, all of them had a response time of less than 1 second. However, if we were looking for variable, which was a much bigger subset, the average response time was about 20 seconds.

6.3.2 Coverage ratio and novelty ratio

From the data of the experiment with human evaluators, we obtained the average coverage ratio and novelty ratio for the 10 sample queries, as shown in Table 6.1.

6.3.2.1 Methods to calculate the ratios

According to the definition of these ratios, we needed to know how many of the results were relevant, how many relevant items were known to the users, and how many relevant items were not retrieved. We obtained these data from the set of ratings about relevance and familiarity given by the participants.

First of all, an item was considered *relevant* if its average relevance ratings was better than a threshold *t* (not including *t*). In other words, only those with moderate or high ratings were relevant results.

Similarly, an item was *known* by the users if its average familiarity ratings was better than the threshold *t* (not including *t*). Otherwise the item was considered unknown by the users.

In order to obtain the above two types of data, we need to set the threshold first. Because the users rated both the relevance and the familiarity from 1 to 5, we set the threshold to be 2, 3, or 4, and could examine the ratios respectively.

	Threshold (*t*)	**Coverage ratio**	**Novelty ratio**
All users	2	87%	65%
	3	25%	97%
	4	0%	100%
Experts	2	69%	48%
	3	50%	76%
	4	25%	92%
Novices	2	100%	85%
	3		100%
	4		100%

Table 6.1 Average coverage ratio and novelty ratio

Besides, in order to calculate how many relevant items known to the users were not retrieved, we asked the users to write down all the missing items that they knew for every query. These items, plus all the retrieved items that they knew were relevant, were all the relevant items known to the user.

Then, we calculated the two ratios for the 10 queries, and got the first set of rows of Table 6.1. We also applied the same methods for experts and novices respectively, and got the remaining rows of the table. Please note that here “experts” refers to those who were very familiar with a query, but were not necessarily those who had worked with the system for a long time. Our rationale was that no one knew every aspect of a large software system, even if he had been with it for more than 10 years; on the other hand, a newcomer could also have lots of knowledge on a specific topic if he had focused on it for weeks or months. In our experiments specifically, if a user’s level of familiarity with a query was more than 2 (not including 2), we regarded this user as an expert on the query; otherwise he was a novice on the query. Therefore, the experts all rated themselves as moderate to high (3, 4, or 5) in terms of their familiarity with the query.

6.3.2.2 Analysis of the ratios

According to the table, our intelligent search tool had high coverage ratio and novelty ratio if the threshold was set to 2, which means that it found most (87%) of the relevant items known by the users, while at the same time brought to them lots of other relevant items (65%) that were previously unknown to them.

The table also showed that the tool was useful not only for novices, but also for experts. For novices who were facing a topic that was new to them, the tool might be a great starting point because it retrieved many results relevant to the topic, among which 85% were previously unknown to them. For experts, although both ratios were lower than that of novices, the tool also covered nearly 70% of what the experts had already known; and among all the relevant results, nearly 50% were new to them.

If the threshold was set to 3, the overall coverage ratio dropped significantly to 25%, while the novelty ratio rose to 97%. If the threshold was 4, the coverage ratio dropped to 0% and the novelty ratio became 100%.

The trend was even clearer for novices, who, according to the above definitions of "experts" and "novices", had little knowledge about the queries. If the threshold was equal to or higher than 3, it was hard to find relevant terms known to them (their level of familiarity with the term was higher than 3), thus we could not compute the coverage ratio because the base was 0. In the other hand, the novelty ratio was 100% because all results were new to them.

Moreover, the effects of the relatively high thresholds on the data of novices also affected the overall coverage ratios and novelty ratios as mentioned above. The reason was that, when the threshold was set equal to or more than 3, only a small (or even zero) portion of the retrieved results were considered *known* to the users as a whole, and the vast majority of the relevant results were new to them.

In the cases of high threshold, it was more reasonable to examine the data from experts only than from all users. Table 6.1 showed that when the threshold went higher, coverage ratio went lower and novelty ratio went higher. In the middle level where the threshold was

3, the ratios showed that half of the relevant items that experts knew were retrieved, while about _ of the retrieved relevant items were new to experts. On the other hand, the low coverage ratio when the threshold was 4 meant that among the relevant items that the experts knew extremely well, the retrieved items were far fewer than those that were not retrieved. Nevertheless, the proportion of the relevant retrieved items that were previously unknown to them turned out to be very high.

In fact, a threshold of 2 was more reasonable than the higher ones for this set of experimental data; because according to the instructions [Appendix B], a rating better than 2 meant that the term was at least moderately relevant to the query. Further investigations showed that it was very harder for the users to reach an agreement in terms of the relevance of the results; it was even hard to find such agreement among experts. Most of the best results presented to the experts that knew them extremely well had an average rating between 3.5 and 4, and quite a few of the results that were considered moderately or highly relevant by those experts could have a rating below 3 (mostly between 2 and 3). Moreover, most of the results that were considered unrelated at all (rated as 1) by the experts had an average rating below 2.

Overall, the coverage ratios and novelty ratios indicated that the tool retrieved a significant portion of relevant items, many of which were new to both experts and novices.

6.3.3 Summary

To sum up, the intelligent search tool did not work very fast, but was much better than normal search tools in satisfying user information needs by finding many relevant items that were previously unknown to users.

6.4 Performance of Candidate Generation Algorithms

Table 6.2 shows for each candidate generation algorithm the number of search candidates derived, the number of results found, and the average ratings of the results provided by our rating algorithms. This table was based on all the 34 queries in our

experiment. Tables 6.3 and 6.4 demonstrate these measures for single-word terms and multi-word terms respectively.

In Table 6.5, we present the distribution of the 178 result terms in the 5 categories according to human evaluation. The sum of the numbers from all the cells of the table is more than 178 because a result may originate from multiple algorithms.

Algorithms	Candidates	Results	Average Rating
ALL	31.61	2307.76	37.27%
ABBR	21.55	1318.88	40.92%
ABBR_ACR	0.97	29.79	61.17%
ABBR_KEY	2.00	68.24	90.00%
ABBR_CUT	20.30	931.94	44.80%
ABBR_SNG	0.97	405.76	45.47%
EXPN	18.53	2242.35	33.69%
EXPN_WHL	1.00	58.80	94.73%
EXPN_CON	1.00	154.65	66.36%
EXPN_VOW	1.00	78.10	84.97%
EXPN_ACR	5.05	109.90	82.00%
EXPN_KEY_CON	6.68	1817.95	33.19%
EXPN_KEY	6.21	750.25	50.46%
KB	1.15	22.24	82.13%

Table 6.2 For each candidate generation algorithm, the number of search candidates derived (Candidates), the number of results found (Results), and the average ratings of the results provided by our rating algorithms (Rating).

6.4.1 Search Candidates generated

From Table 6.2, we can see that on average, the algorithms altogether derived more than 30 search candidates for each query. Step 3 (ABBR), which derived the abbreviation format of the original string, found about 2/3 of the search candidates alone. However, in single-word term cases it derived less than 1/2 of the candidates, whereas in the multi-word

cases it generated more than 90% of them. For both query term types, concatenating the abbreviations of the keywords (ABBR_CUT) derived the highest number of candidates among all algorithms.

Algorithms	Candidates	Results	Average Rating
ALL	30.89	2879.10	32.88%
ABBR	14.79	1198.00	39.08%
ABBR_ACR	0.00	0.00	
ABBR_KEY	2.00	61.85	93.11%
ABBR_CUT	13.68	1199.60	39.11%
ABBR_SNG	0.00	0.00	
EXPN	18.53	2242.35	33.69%
EXPN_WHL	1.00	58.80	94.73%
EXPN_CON	1.00	154.65	66.36%
EXPN_VOW	1.00	78.10	84.97%
EXPN_ACR	5.05	109.90	82.00%
EXPN_KEY_CON	6.68	1817.95	33.19%
EXPN_KEY	6.21	750.25	50.46%
KB	0.63	12.80	87.83%

Table 6.3 For each candidate generation algorithm in single-word term cases, the number of search candidates derived (Candidates), the number of results found (Results), and the average ratings of the results provided by our rating algorithms (Rating).

Step 4 (EXPN), which was designed solely for the single-word cases, on average accounted for more than a half of the candidates alone. Most of the candidates were formed by decomposing the query strings using the dictionary of abbreviations (EXPN_ACR), or taking the unique parts of the decomposed query strings (EXPN_KEY and EXPN_KEY_CON).

Algorithms	Candidates	Results	Average Rating
ALL	32.57	1491.57	43.55%
ABBR	30.71	1491.57	43.55%
ABBR_ACR	2.29	72.36	61.17%
ABBR_KEY	2.00	77.36	86.18%
ABBR_CUT	29.29	549.57	52.93%
ABBR_SNG	2.29	985.43	45.47%
KB	1.86	35.71	65.00%

Table 6.4 For each candidate generation algorithm in multi-word term cases, the number of search candidates derived (Candidates), the number of results found (Results), and the average ratings of the results provided by our rating algorithms (Rating).

The Knowledge Base (KB) helped us find about 1 search candidate per query on average. We noticed that it found nearly two search candidates for multi-word terms, while less than one candidate for single-word terms. Maybe this is because the concepts in the knowledge base mostly consisted of English words rather than abbreviations, and the abbreviations were mostly defined in the dictionary of abbreviations.

To sum up, ABBR and EXPN produced more than 90% of the candidates. The biggest contributor among the sub-algorithms was ABBR_CUT, and each of EXPN_ACR, EXPN_KEY, and EXPN_KEY_CON also found more than five candidates.

6.4.2 Quantity of results found

In our experiments, a multi-word term on average produced nearly 1500 results, while a single-word term found nearly 3000. No matter which case a human is facing, it is infeasible for him to deal with every item of such a big group. We therefore have to investigate the effectiveness of each algorithm, as we will explain later in this chapter, so that we can adjust the algorithms and find out the best way of removing bad results from the query return.

From Table 6.2, we notice that in general, big contributors of search candidates also found lots of results. For example, each of ABBR_CUT, EXPN_ACR, EXPN_KEY and EXPN_KEY_CON alone could find hundreds or even more than a thousand results on average.

However, even though some algorithms did not compute many search candidates, they queried a large number of results. For instance, ABBR_SNG, which retrieved every single word in the multi-word term cases, generated nearly 1000 results. In addition, EXPN_CON computed only one candidate, but queried about 150 results every time.

It should be also noted that in single-word cases, although EXPN_ACR computed about the same number of candidates as did EXPN_KEY and EXPN_KEY_CON, it obtained slightly more than 100 results, which were many fewer than that of the latter two algorithms.

The remaining algorithms, which are ABBR_ACR, ABBR_KEY, EXPN_WHL, EXPN_VOW, and KB, found less than 100 results per query.

In short, we may say that the shorter versions of the query strings, which were derived from ABBR_CUT, ABBR_SNG, EXPN_KEY and EXPN_KEY_CON are the major contributors of the query results.

6.4.3 Quality of the results

6.4.3.1 Automatic evaluation

The quality was first measured by the average rating, which was the mean value of the ratings of all the results from an algorithm. High average rating of an algorithm should indicate that its overall findings are highly relevant. As Table 6.2 shows, in general, results from algorithms that queried hundreds or even thousands of terms had less relevancy than results from the rest of the algorithms.

1) Results from ABBR. Among the abbreviation concatenation methods (ABBR), the first two algorithms (ABBR_ACR and ABBR_KEY) found results with moderate to high ratings, while ABBR_CUT and ABBR_SNG had average ratings around 45%.

The acronyms of the original query string (ABBR_ACR), which were for multi-word cases exclusively, found quite a few results generally, and sometimes even contributed to 90% of the results. However, their findings had only moderate ratings (around 60%), even if at many times these findings are very useful results. This is reasonable, though, because the acronyms' *likelihood of appearance* in the results might be very low according to our rating algorithms. Since acronyms were generally short (3 or 4 characters long), sometimes they introduced a large number of obviously unrelated results, along with many highly relevant ones, that contained an English word that had the acronyms as one of its sub-strings. These “bad” findings were rated as low as 40%, and therefore largely hurt the performance of the acronym algorithms.

Further investigations revealed that most of the bad findings from ABBR_CUT were from the candidates that took only the first 2 or 3 letters of the keywords as their abbreviations. However, considering the fact that these candidates also contributed lots of good results, we did not want to discard them, lowering the recall, but rather to rely on the rating algorithms to improve the precision by identifying and removing the bad results.

The results also supported our selection of the lengths of the abbreviations (2 to 6) in ABBR_CUT. Shorter abbreviations (containing 2 or 3 letters) found a significant number of results, while longer ones found highly relevant results. In fact, in most cases, abbreviations with 5 letters long retrieved the results that were almost identical to the query. Thus, if we want to improve the performance of the query, we can change the ABBR_CUT algorithm by computing abbreviations with a length from 2 to 5, which reduces the number of candidates to compute and query without losing any good results.

2) Results from EXPN. Among the abbreviation expansion methods (EXPN), querying the whole strings (EXPN_WHL), splitting the abbreviations before vowels (EXPN_VOW), or splitting them using the abbreviations from the dictionary of abbreviations (EXPN_ACR) found closely related results. Considering the fact that EXPN_ACR also provided more candidates than other abbreviation expansion methods, we can say that the dictionary of abbreviations was the most important source of intelligence in the single-word cases.

Besides the above methods, expanding the abbreviations after every consonant provided moderate returns. And querying parts of the original strings (EXPN_KEY and EXPN_KEY_CON) obtained a large number of poor results.

3) Results from KB. Finally, although the results from the knowledge bases (KB) only made a very small portion of the whole retrieved set, they were all highly rated.

Algorithms	**C1**	**C2**	**C3**	**C4**	**C5**
ABBR					
ABBR_ACR	0	2	1	4	0
ABBR_KEY	2	8	4	18	3
ABBR_CUT	41	32	21	29	5
ABBR_SNG	7	10	3	5	0
EXPN					
EXPN_WHL	2	4	1	14	3
EXPN_CON	11	0	2	5	1
EXPN_VOW	1	3	0	4	1
EXPN_ACR	6	3	0	4	1
EXPN_KEY_CON	14	13	7	2	0
EXPN_KEY	11	7	8	1	0
KB	0	2	1	5	1

Table 6.5 Number of results generated by each algorithm in the 5 human rating categories (C1 to C5).

6.4.3.2 Human evaluation

Our analysis of the user ratings also verifies most of the above observations. In Table 6.5, we categorized the 178 results evaluated by the software engineers from 1 to 5, according to their average human ratings. We again noticed that the worst results (rated 1 or 2), which were also the biggest portion of the whole result set, were mainly from truncating the original query string (ABBR_CUT), searching the parts of the query string (ABBR_SNG, EXPN_KEY, EXPN_KEY_CON), and decomposing the single-word strings after all consonants (EXPN_CON). Other methods found fewer, but gave fairly good results (rated better than 3).

6.4.4 Summary of the performance of the candidate generation algorithms

According to the above discussion, the performance of the candidate generation algorithms in terms of the quality of the results can be cast into three categories:

- High: ABBR_KEY, EXPN_WHL, EXPN_VOW, EXPN_ACR and KB were the algorithms whose results have an average rating of more than 80%.
- Medium: ABBR_ACR and EXPN_CON found results with their average ratings of more than 60%.
- Low: ABBR_CUT, ABBR_SNG, EXPN_KEY, EXPN_KEY_CON deluged users with poor findings.

6.5 RATING ALGORITHMS

We assessed our rating algorithms by comparing the ratings given by them to ratings given by human evaluators. We first performed some case studies to see if the rating algorithms judged the results in a reasonable and consistent way. Table 5.2 and 5.3 were two samples among those we had investigated.

Next, we examined the accuracy of the rating algorithms by comparing their values with human average ratings for the 178 results. The summaries are shown in tables 6.6 to 6.10. Tables 6.6, 6.7 and 6.8 shows the percentage of results handled by each algorithm, and their average ratings, as computed by our automatic algorithms. Table 6.10 demonstrates for the five human rating categories, the average ratings given by the rating algorithms, along with the standard deviation, and median, minimum, and maximum values. Finally, Table 6.9 shows for each rating algorithm its average accuracy by comparing their ratings with 20 times the average human evaluations.

Last we performed correlation coefficient analysis between different sets of ratings.

6.5.1 Evaluation of the accuracy from case studies

From the two samples given in Table 5.2 and 5.3, we notice that the ratings given by the algorithms were reasonable and consistent. Most of the highly rated results were considered

highly relevant by human, while most of the poorly rated ones were regarded as irrelevant terms. In other words, many of the ratings fell into the same range as that of the human ratings.

In addition, ranking the terms according to their ratings from high to low provided mostly the same order as that given by ranking them according to the software engineers' evaluations. Consequently, even if their individual ratings given by the algorithms are not accurate, the results presented in descending order according to their ratings should be able to promote the best ones to the top of the list, and hence should increase the chances for users to investigate the most relevant results.

Moreover, the best ratings were given by Algorithm 1, which used related terms as the evaluation string. Next were ratings from the second algorithm, which handled the majority of the returns. Algorithm 3 covered the remaining, mostly poor results.

Noticeably, the rating algorithms had difficulty in assessing some types of results. For example, in Table .5.3, "list*debug" can easily be judged by humans as highly relevant to "listdbg", but the rating algorithm only gave it a moderate rating value. However, this is the hardest part of the rating algorithm. Our methods at least made "list*debug" about 15 points superior to other "list*"-like terms by identifying that "debug" might be a good representative of "dbg".

6.5.2 Average accuracy of the rating algorithms

6.5.2.1 Automatic evaluations

Table 6.6 tells us that in the 34 queries, Algorithm 1 handled less than 4% of the results, but these were the best results. Algorithm 3 handled about 10%, but they were most noisy. Algorithm 2 covered more than 80% of all the results, but the average rating was very low (less than 40%). This could mean that the rating algorithm underestimated the results, or there was too much noise in the results.

Rating Algorithms	Percentage Rated	Average Rating
R1	3.61%	87.60%
R2	86.55%	37.68%
R3	9.84%	20.00%

Table 6.6 Percentage of all results handled by the rating algorithms and their average ratings.

Rating Algorithms	Percentage Rated	Average Rating
R1	2.75%	93.72%
R2	80.52%	34.11%
R3	16.73%	20.00%

Table 6.7 Percentage of results handled by the rating algorithms and their average ratings in single-word cases

Rating Algorithms	Percentage Rated	Average Rating
R1	4.84%	78.93%
R2	95.16%	42.78%
R3	0.00%	

Table 6.8 Percentage of results handled by the rating algorithms and their average ratings in multi-word cases

	Mean	Std Dev	Median	Min	Max
Algorithm 1	1.19	0.25	1.12	0.94	2.03
Algorithm 2	1.13	0.48	1.06	0.34	2.88
Algorithm 3	0.85	0.21	1.00	0.29	1.00

Table 6.9 Average accuracy of the rating algorithms (average ratings from the algorithm/(the average human rating * 20))

	C1	C2	C3	C4	C5
Mean	34.41	45.43	49.53	76.47	96.39
Std Dev	11.92	16.29	21.18	22.61	2.31
Median	31.86	42.50	51.12	91.36	95.96
Min	20.00	23.21	20.00	32.50	93.64
Max	63.13	91.20	91.50	99.30	100.00

Table 6.10 Summary statistics: the ratings by our rating algorithms in the 5 human rating categories (C1 to C5).

From Table 6.9, we in fact see that except for Algorithm 3, which gave the lowest possible rating to every result, other algorithms slightly overestimated the results. Algorithm 1 outperformed Algorithm 2 by rating the best results while with lower standard deviation. However, as we have discussed above, Algorithm 2 was doing an intrinsically harder job.

In addition, Table 6.7 and Table 6.8 show that Algorithm 1 handled fewer results in single-word cases than it did in multi-word cases, but the average ratings were higher. This was because single-word terms contained only one word in the query, therefore, it was easier for the first algorithm to judge the results.

Algorithm 2 assessed a larger proportion of the results in multi-word cases than in single-word cases, yet the average ratings were better. Maybe this was because the word delimiters in multi-word queries helped the algorithms accurately decompose the strings for both query expansion and result evaluation.

Algorithm 3 handled more than 16% of all results in single-word cases, but none in multi-word cases. The reason was that the results left to this algorithm were mostly retrieved by querying search candidates like "*d*b*g*", which were impossible to generate from our rating algorithms for multi-word cases.

6.5.2.2 Human evaluations

Another good way to assess the performance of the rating algorithms in general is to compare their ratings with human evaluation. We can develop some patterns of our

algorithms from Table 6.10, which gives the ratings by our algorithms in the 5 user rating categories:

- The mean ratings of category 1, 2, 3, 4 and 5 were roughly at a level of 34%, 45%, 50%, 76% and 96%, respectively. The ideal average rating should be 20%, 40%, 60%, 80%, and 100%.
- The rating algorithms were strong at identifying highly relevant results (rated by users as 4 or 5) or the mostly irrelevant ones (rated as 1) from the others.
- It was hard for our rating algorithms to draw a fine line between the results with moderate relevance (rated by human as 3) and those with low relevance (rated as 2).

To sum up, our algorithms provided reasonable ratings for the results, especially for the best. Typically, the higher the rating, the better the accuracy. Low accuracy mainly occurred at the moderate to low rating levels. However, our algorithms were good at identifying the worst results, too, especially those rated by Algorithm 3. If the poor results identified by our algorithms were pruned from the return, the overall precision should have obvious improvement.

6.5.2.3 Percentage of false positives

Another way to judge the rating algorithms is by examining the number of false positives. False positives are the items that are considered relevant by the tool, but irrelevant by users.

	Average user rating <		
Rating given by the algorithms >	**2**	**3**	**4**
60%	3.09%	8.02%	17.28%
70%	2.47%	4.94%	10.49%
80%	2.47%	3.09%	5.56%

Table 6.11 Percentage of false positives using different thresholds

A result with a rating of 60% by our rating algorithms was considered fairly close to the original query. Thus, it was reasonable for us to set 60% as a threshold to judge the relevance of the results. Any result with a rating better than 60% was relevant.

Again, the threshold to judge user opinion could be set to 2, 3, or 4. If the threshold of user rating was 2, then the percentage of false positives was the proportion of relevant results according to the rating algorithms (the ratings by the algorithms were better than 60%) that were considered irrelevant by users (user ratings worse than or equal to 2). As shown by the first column in the first row of Table 6.11, about 3% of the results were false positives. Similarly, other cells of the table illustrated the percentage false positives using different thresholds.

From Table 6.11, it was safe to say that more than 80% of the results that had a rating better than 60% were relevant to the query. Moreover, among the results that were rated at least 80%, only less than 6% of them were considered irrelevant. Hence, the proportion of noise among the relevant results was very low.

6.5.3 Correlation coefficient analysis

Finally, we need to decide the reliability of our experimental data, because we were using human evaluators, who in general are not always consistent when evaluating the same thing. In statistics, people usually use correlation coefficients to judge the inter-rater reliability [26]. We, too, used the most common Pearson's correlation [9] to assess the reliability or consistency of our data.

Pearson's correlation measures the degree of the relevance of two sets of data by giving a number between -1 and 1. A correlation of 0 means the two sets of data have no relationship at all, while 1 means they have a perfect positive linear relationship. In our case, a correlation of 1 means the two raters have a maximum level of agreement, and therefore, given the ratings from one of the two raters, it is very easy to predict that of the other.

Table 6.12 lists some correlation we computed, which are explained below:

- **Average inter-rater correlation**. We first calculated the correlation between the ratings of every two users, and then took the average of the correlation values. 0.64 showed that users did not have a high level of agreement with each other in rating the results. Their difference in background, experience, domain knowledge, work requirement, and many other aspects determined that they could interpret the same result in very different way.

  This poor average correlation may imply that we do not have to seek perfect consistency between our rating algorithms and human evaluators, because sometimes humans do not agree with each other, either.

  Moreover, we believe our previous judgments based on average human ratings are still useful because the average values should have minimized the differences among people and have shown their commonness.

Groups to compare	**Correlation coefficient**
Average inter-rater correlation	0.64
Average rater-average correlation	0.86
Experts vs. novices	0.83
Rating algorithms vs. all users	0.71
Rating algorithms vs. experts	0.66
Rating algorithms vs. novices	0.70

Table 6.12 Correlation coefficient between different sets of ratings

- **Average rater-average correlation**. We calculated the correlation coefficient between the average rating and that of every user, then took the average of the correlation coefficients. This value, 0.86, was better than the average inter-rater correlation because everybody's rating had contributed a part to the average ratings.
- **Experts vs. novices**. We also computed the average ratings of experts and novices. The correlation coefficient between these two sets of values was 0.83, which means there were no big difference between the judgment of experts and novices of the system.
- **Rating algorithms vs. all users**. When comparing the ratings given by the rating algorithms with the average ratings given by all users, the correlation coefficient

was 0.71. Interestingly, this number was better than the inter-rater correlations. As discussed above, although 0.71 did not show a perfect linear relationship between the two sets of data, it was a good number that revealed the relatively consistent relationship between our rating algorithms and human evaluators as a whole.

- **Rating algorithms vs. experts**, and **Rating algorithms vs. novices**. We also compared the ratings given by the rating algorithms with the average ratings given by experts of the systems, as well as with that given by novices. There was no big difference between these two correlation values. However, it seemed that the rating algorithms were slightly closer to the attitudes of novices than that of experts. Maybe this means both our rating algorithms and novices lacked the domain knowledge that experts had.

To sum up, the correlation coefficient analysis showed that although every individual user might rate a result differently, there was in general no big difference between the ratings given by novices and experts. It also showed that the ratings computed by our rating algorithms were a fair representative of average human ratings, and hence our evaluation of the candidate generation algorithms using the data from our rating algorithms was reasonable.

6.6 IMPROVING THE TOOL

From the above experiment analysis, we can see that the intelligent search tool was effective in finding relevant results, and the rating algorithms were reasonably good at evaluating the findings. Therefore there is no urgent need for us to significantly change the text operations and the initial settings in the rating algorithms at the moment.

The biggest problem was that the response time was long. According to previous analysis, this problem was caused by retrieving all the generated search candidates one by one in a form of regular expressions. If we can manage to reduce the number of search candidates, the response time can also be shortened.

6.6.1 Methods

The principle of the improvement strategy was to reduce the search candidates that can find duplicated results as much as possible. In the following, we describe in detail how we improved the tool.

As mentioned before, a result can be found by multiple search candidates. If we can reduce the number of the candidates that find the same results, the retrieval should take less time without losing any results.

ABBR_CUT, the candidate generation algorithm that computed abbreviations through truncation, produced the most search candidates among all algorithms. However, many of these search candidates found the same results; that is, the results found by a longer version of abbreviations were usually covered by those retrieved by a shorter one. For example, all the results found by retrieving "*busy*" could be found by retrieving "*bus*". Hence, we reduced the longer versions of the abbreviations and retained only the ones with a length of 2, and the tool had many fewer candidates to search, yet found the same number of results.

Other big contributors of search candidates were the EXPN_ACR, EXPN_KEY, and EXPN_KEY_CON algorithms. EXPN_KEY and EXPN_KEY_CON each found a large number of results, most of which were noise. Moreover, the results found by EXPN_KEY could all be found by EXPN_KEY_CON. For example, the results found by querying "*dbg*" (from EXPN_KEY) can be found by querying "*d*b*g*" (from EXPN_KEY_CON). In order to improve the performance, we should drop the search candidates from one of these two algorithms. According to the previous experiments, EXPN_KEY_CON could find far more results than EXPN_KEY, but the average rating of its results was very poor, only 33%. Considering the fact that retrieving the search candidates from EXPN_KEY_CON, such as "*d*b*g*", was very inefficient, we decided to drop these candidates. In this way, we lost some results, but they probably were mostly the "bad" results that usually had little chance of being examined by the users, even if they were retrieved.

EXPN_ACR also generated many search candidates. Each represented one way to decompose the single-word query string. Although normally only one of the decompositions was correct, the algorithm did not tell us which one was. Thus the tool's

performance was hurt by having to try all the possibilities. Yet we would not drop these candidates because they were not replaceable by the others.

Noticeably, no matter what search candidates we dropped, the process of computing the candidates remained unchanged. Thus, the candidate generation algorithms produced the same set of evaluation strings as they did before the improvement. Consequently, the rating algorithms would also give the same rating to a result as they did before.

6.6.2 Improved performance

Through the above methods, the average response time was improved to less than 20 seconds, if the user wanted to retrieve all types of objects in the database. When they only retrieved variables, which were the biggest subset of the database, the average response time was within 10 seconds. Retrieving other types, such as routines, took less than 5 seconds.

6.7 LIMITATION OF THE HUMAN EXPERIMENT

The major drawback of the human experiment was that it was using prerecorded queries, rather than the live queries that the participants posed while actually using the tool. In this case, the participants only judged the results by comparing them with general information needs, without knowing the *context* (i.e. the original intent of the query). However, the data of the experiments were still valuable, because:

- They show that the tool can retrieve a set of results that would be relevant in general. Results relevant to a specific context would most likely be a subset of these, so the tool would find those, too.
- The experiments helped us identify the strength and weaknesses of the tool. Hence, we could improve it according to the data from these experiments before we perform further studies. We leave it as future research to conduct a series of other experiments, including the evaluation of search results for live queries posed by human developers.

6.8 SUMMARY OF THE EXPERIMENT ANALYSIS

From the experiment analysis, we learned that in general:

- The intelligent search tool was effective at retrieving relevant results in large software systems.
- Its response time could be reduced to an acceptable level, but in general the tool was not as fast as normal information retrieval systems because it had no such "inverted-file"-like indexing mechanisms as those systems had.

As for the search candidate algorithms:

- The knowledge base helped the tool find many relevant results with only a few search candidates.
- The dictionary of abbreviations was very useful for decomposing the single-word query strings and retrieving many relevant results, but also hurt the performance by suggesting many different ways of doing decomposition.
- Search candidates representing the whole query string found fewer, but more relevant results than those representing only a part of the query string.

As for the rating algorithms:

- The rating algorithms gave reasonable ratings to search results.
- The rating algorithms were good at identifying the most relevant and the irrelevant results.
- The rating algorithms were relatively weak at evaluating results with moderate to low relevance.

CHAPTER 7. CONCLUSION AND FUTURE WORK

7.1 REVIEW OF THE RESEARCH

This thesis has presented our experience of using intelligent search techniques in a large software system. Our methods are relatively simple to implement, and are shown to be helpful in finding relevant entities for the queries in software maintenance in our preliminary experiments.

We first selected a model for the intelligent search tool that was suitable for software systems and was relatively easy to implement. The model was designed in such a way that it took the advantage of the intelligent search methods, as well as of the existing DBMS query mechanisms.

Then, we selected a group of text operations that were suitable for software maintenance. After that, we applied these techniques in the search candidate generation algorithms to expand the original query. The algorithms generated many search candidates, and each of the candidates retrieved many results through the query mechanisms.

Next, we designed several rating algorithms to automatically evaluate and rank the query results before they were presented.

We also performed a series of experiments to analyze the efficiency and the effectiveness of the tool. We showed that this was a useful tool for software maintenance.

Finally, according to the experiment results, we slightly adjusted the search strategies to improve the performance of intelligent search.

7.2 CONCLUSIONS

Through this study, we learned that the wide spread usage of abbreviations was the major factor that differentiated source code systems from other documents in terms of information retrieval. This difference leads to the observation that many information retrieval techniques that worked well in normal systems were not appropriate for software systems.

When expanding the queries, the most important techniques we used were abbreviation concatenation and abbreviation expansion. Among these methods, taking the acronyms, removing the vowels, and truncating the query were all very useful for finding relevant results.

In addition, the use of domain knowledge also played an important role in retrieving and evaluating the results. Usually, the domain knowledge that could be used by a tool was in a form of knowledge base (the dictionary of abbreviations we used could also be treated as one type of knowledge base). However, these knowledge bases had to be constructed before being applied in the intelligent search, and doing so required considerable manual effort. However, as was pointed out in [22], "a little knowledge goes a long way". In other words, a relatively simple knowledge base can be of considerable use.

In the rating algorithms, the usage of evaluation strings made the judgment of the similarity between strings that involved abbreviations much easier. The ratings of the results were close to those given by human evaluators, and the order of the results ranked according to the ratings computed by the algorithms was very similar to that provided by the users.

7.3 Limitations and Future Work

The major limitation of the intelligent search tool was that the response time was relatively long. However, within the current model it was very hard to ultimately solve this problem. One way for the tool to reach the speed that normal information retrieval systems have, is to build an index that lists all the fragments of the identifiers found in the system. Future research should study how such an index can be constructed automatically. For example, we could use the text operations described in this thesis to decompose all identifiers in a system during the pre-processing stage. Although these text operations will suggest multiple decomposition methods for every identifier, they are much cheaper to adopt than those involving human beings.

In the candidate generation algorithms, there are other text operations that have not been used; therefore their usefulness is not entirely clear. For example, further studies should show to what extent stemming, soundex, and antonyms add value to the intelligent

search tool in software systems. Moreover, when investigating the soundex techniques, there should be some research on the specific soundex algorithm for software systems.

Another weakness of our methods was that they only minimally used domain knowledge. The only domain knowledge we used were from some pre-built knowledge bases, without which the candidate generation algorithms and the rating algorithms would be less effective. Future research should study how to obtain the domain knowledge automatically in the absence of the knowledge bases. For example, getting the term-term relationship from the context, the comments, and the reference information in the source code is one possible approach.

Further studies should also investigate the general effectiveness of intelligent search for software maintenance. The effectiveness could be measured in many different aspects, for example, the general usefulness of the intelligent search techniques in solving the software engineers' problems, the time saved by the software engineers when they use a tool with intelligent search, or the usability of such tools. The assessment of the effectiveness involves more experiments with the software developers as they are actually maintaining a large software system.

Finally, the intelligent search techniques should be generalized in such a way that not only does it work well with TkSee within a specific software domain, it can also be part of other tools and be efficient in other software systems. For instance, the intelligent search tool should easily incorporate other knowledge bases, or be incorporated into any search tool. Future study should also examine how these techniques can benefit searches in places other than software systems, e.g. normal document systems or information resources on the Internet.

REFERENCES

1. A list of English stopwords, http://oradoc.photo.net/ora81/DOC/inter.815/a67843/astopsup.htm

2. Albrechtsen, H., “Software information systems: information retrieval techniques”. In Hall, P., *Software Reuse and Reverse Engineering*, Chapman & Hall, London, New York, 1992.

3. Anquetil, N. and Lethbridge, T., "Extracting concepts from file names; a new file clustering criterion", in *International Conference on Software Engineering ICSE'98*, pages 84-93. IEEE, IEEE Comp. Soc. Press, Apr. 1998.

4. Baeza-Yates, R. "Introduction to Data Structures and Algorithms related to Information Retrieval". In W. Frakes and R. Baeza-Yates, editors, *Information Retrieval: Data Structure and Algorithms*, Prentice hall, Englewood, NJ, 1992.

5. Baeza-Yates, R. *Modern information retrieval*, Addison-Wesley, New York, 1999.

6. Boldyreff, C. (1989), “Reuse, Software concepts, Descriptive methods and the Practitioner project”, in *ACM SIGSOFT Software Engineering Notes* 14 (2), pp. 25-31.

7. Bolstad, J. (1975), “A proposed classification scheme for computer program libraries”, in *SIGNUM Newsletter* 10 (2-3), pp. 32-39.

8. Brin, S. and Page, L. (1998), "The anatomy of a large-scale hypertextual web search engine", In *Proceedings of the 7th International World Wide Web Conference*, pages 107--117, Brisbane, Australia, April 1998. Elsevier Science. Online source: http://www7.scu.edu.au/programme/fullpapers/1921/com1921.htm.

9. Computing Pearson's correlation coefficient. http://davidmlane.com/hyperstat/A51911.html.

10. Frakes, W.B.; Nehmeh, B.A. (1987), “Software reuse through information retrieval”, in *SIGIR FORUM* 21 (1-2), pp. 30-36.

11. Korfhage, Robert R. "Research in Relevance Feedback and Query Modification". In *Information Storage and Retrieval. New York*: John Wiley & Sons, Inc.

12. Lethbridge, T. and Anquetil, N., (1997), "Architecture of a Source Code Exploration Tool: A Software Engineering Case Study", University of Ottawa, *CS Technical report TR-97-07*. http://www.site.uottawa.ca/~tcl/papers/Cascon/TR-97-07.html

13. Lethbridge, T., "Integrated Personal Work Management in the TkSee Software Exploration Tool", *Second International Symposium on Constructing Software Engineering Tools (CoSET2000),* ICSE 2000, pp. 31-38.

14. Lethbridge, T. and Singer Janice, "Studies of the Work Practices of Software Engineers", In *Advances in Software Engineering: Comprehension, Evaluation, and Evolution*, H. Erdogmus and O. Tanir (Eds.), Springer-Verlag, pp. 53-76.

15. List of software visualization tools, http://www.iro.umontreal.ca/labs/gelo/sv-survey/list-of-tools.html.

16. Maletic, J. I. and Marcus, A., "Supporting Program Comprehension Using Semantic and Structural Information", in *Proceedings of 23rd ICSE*, Toronto, 2001, pp. 103-112.

17. Miller, G. (1990), "WordNet: An On-line Lexical Database", in *International Journal of Lexicography*, vol. 3, No. 4, 1990.

18. Morris, J. and Hirst, G. (1991), "Lexical cohesion computed by thesaural relations as an indicator of the structure of text", in *Computational Linguistics*, 17:21-48.

19. Microsoft Visual C++, http://msdn.microsoft.com/visualc.

20. Roget's thesaurus, http://www.soi.city.ac.uk/text/roget/thesaurus.html.

21. Salton, G. and Buckley, C., "Improving Retrieval Performance by Relevance Feedback", in *Journal of the American Society for Information Science*, 41(4):288-197, John Wiley & Sons, 1990.

22. Sayyad-Shirabad, J., Lethbridge, T. and Lyon, S. (1997), "A Little Knowledge Can Go a Long Way Towards Program Understanding", *IWPC*, Dearborn, MI., pp. 111-117.

23. SHriMP (University of Victoria), http://www.csr.uvic.ca/~mstorey/research/shrimp/shrimp.html.

24. The Soundex Indexing System, http://www.nara.gov/genealogy/coding.html.

25. Source Navigator, http://sources.redhat.com/sourcenav/index.html.

26. Types of Reliability. http://trochim.human.cornell.edu/kb/reltypes.htm.

27. Wood, M.; Sommerville, I. (1988), “An Information Retrieval System for Software Components”, In *SIGIR FORUM* 22 (3-4), pp. 11-25.

28. WordNet, http://www.cogsci.princeton.edu/~wn/.

29. Su, L. T. (1994), “The Relevance of Recall and Precision in User Evaluation”, in *Journal of the American Society For Information Science (JASIS)*, 45(3), pp.207-217, 1994.

30. IBM Java Technology, http://www-106.ibm.com/developerworks/java/.

31. **jCentral**: The Ultimate Resource for Java, Developershttp://ftp.mayn.de/pub/unix/devel/MIRROR.docs/jcentral.pdf.

APPENDICES

APPENDIX A: SAMPLE QUESTIONAIRE OF THE HUMAN EXPERIMENT

1. Query string: “CallForwardWhileBusy”

How familiar are you with the query string? _____

1 = no idea at all about it

2 = know a little

3 = know it moderately

4 = know it well

5 = extremely familiar with it

Results:

Results:	How useful is the result?	How interesting is the result?	How familiar are you with the result?
Result 1			
Result 2			
Result 3			
…			

Please list any other relevant strings that you know, not found by the query:

APPENDIX B: INSTRUCTIONS FOR THE HUMAN EXPERIMENT

Evaluating Intelligent Search Results – Instructions

Purpose:

To evaluate the algorithms of, and identify problems in the Intelligent Search function which will be added to the software system entitled "TkSee".

Intelligent Search is a technique used when exact search does not find the expected results. Intelligent Search expands the scope of the search by loosening search conditions; for example, allowing parts of the query string to be missing in the target, allowing different formats of the query string in the target, searching for synonyms, and so on.

The Intelligent Search function in "TkSee" aims at helping software developers work more efficiently with a legacy system. For novices of the legacy system, Intelligent Search may help them figure out what names are used to represent a certain concept and where those names are used. For developers who have lots of knowledge of the system, this tool will help them find out interesting objects which are related to the expected target.

The purpose of this experiment, therefore, is to examine how well this tool is designed and how much the future users will be benefit from it. The result of this experiment will be used to fine-tune the tool and to gather general scientific information that will be of use to designers of similar tools.

Tasks:

You will be asked to evaluate the results of 15 Intelligent Search queries. For each query, you will be given an input string, which is the string the Intelligent Search function was asked to search for. And you will be given about 10 to 20 result items to evaluate. These result items were selected randomly from the results of each query, and were not sorted in any sense. The results are *not* those you will see in the final tool, since in the final tool you will only see the best results.

In the sample below, we will explain each question you will be asked to answer.

Imagine you were searching for '*CallForwardWhileBusy*'. You are, first of all, asked about your knowledge of this string:

· *How familiar you with the query string?* (CallForwardWhileBusy in this example – A query string is familiar to you if you have heard of it, or have worked on it, or you are confident you can guess its meaning)

You will be asked to choose from one of the following options.

1 = no idea at all about it
2 = know a little
3 = know it moderately
4 = know it well
5 = extremely familiar with it

You will then be given some of the results generated by the search engine. You will be asked to answer the some questions about each result, on a scale of 1 to 5. The questions will appear in a table that looks like the following:

Results:	How useful is the result?	How interesting is the result?	How familiar are you with the result?
Result1			
Result2			

You fill in blank cells in the above table, as follows:

· *How useful is the result?* (A result is useful if you believe it is what you would be looking for)

1 = not useful at all
2 = a little useful
3 = moderately useful

4 = very useful

5 = exactly what I want

· *How interesting is the result?* (A result is interesting if it reveals something new to you even it is not useful right now, or if you think there is a meaningful relationship between the query and the result, but you would not have easily known about or found this result yourself)

1 = uninteresting

2 = a little interesting

3 = moderately interesting

4 = very interesting

5 = extremely interesting

· *How familiar are you with the result?* (A result is familiar to you if you have heard of it, or have worked on it, or you are confident you can guess its meaning)

1 = no idea at all about it

2 = know a little

3 = know it moderately

4 = know it well

5 = extremely familiar with it

· *There are other relevant strings that I know, but are not found by the query (please list):*(In this example, suppose you know that "cfb" stands for "call forward busy" in this system, and you believe it is relevant to the query string, then you may write it down as)

cfb, or

cfb

After you finish all the queries, you will then be asked to answer some general questions.