An adaptive intranet search engine

Colin Cherry, Shawn Wallace, André Trudel
Intelligent Information Technology Research Centre
Jodrey School of Computer Science
Acadia University
Wolfville, Nova Scotia
B0P 1X0, Canada

Abstract

This paper presents a search engine that can be used to search a specially index ed collection of documents. This search engine is designed to be intelligent. I t keeps track of all successful queries and the frequency of record and keyword usage. Using this information, the engine improves its accuracy and performance over time.

1 Introduction

Our target application domain is a call center. At a call center, it is the responsibility of customer service representatives (CSRs) to quickly and accurately answer questions they receive. To help them with this task the CSRs are supplied with a very large database of helpful information. This database should be searched in such a way that the following requirements are met:

1. The list of possible results for a given query must be returned instantly. A customer is waiting on the other end of the line for the CSR's response to his or her question.

2. The results of the search must be displayed to the CSR so that the correct choice among them can be determined quickly.

3. Regardless of which CSR takes the call, customers should be given consistent replies.

The search engine presented in this paper is designed to meet all of the requirements listed above, thus maximizing the efficiency of the call center scenario.

This search engine grows each time it is used. It is designed to store information, not only on the documents it has indexed, but also on the searches it has made in the past. Once given a query, the engine returns a list of similar queries it has answered in the past.

If the user judges one of these past queries to be close enough to that which he or she was searching, then he or she is taken directly to the correct answer to that question. If the user decides that none of the past queries match that which he or she is searching, a ranked list of documents that may contain the answer to the query is returned. A list of related keywords is also presented which can be used to refine or broaden the search quickly. Once the correct answer to the query is found, the engine updates its own indexes by storing the query, its keywords, and its correct answer. These updates may be used to modify future responses.

This paper describes the tables used for indexing the search engine, the search algorithm, the current state of the implementation, and a list of suggested improvements.

2 Table Descriptions

Various tables, stored in a relational database, are required by the search engine:

2.1 keywords

Each word in each document to be searched is a possible keyword. To maximize the efficiency of the search the number of keywords has to be minimized. This is accomplished by applying a number of rules to each potential keyword to determine whether or not it is valid. First, all words that are only one character long and words that consist only of numeric characters are removed. In the next step, plural words have been combined with their singular counterparts by removing 's' suffixes. Because the bulk of information being searched is in French, feminine words are combined with their masculine equivalents by removing 'e' suffixes. Next, each word is compared to the stoplist described below. It is removed if it matches an entry on the stoplist. Lastly, a count of all remaining words is maintained and all words that occur less than 5 times or more than 300 times throughout the database are removed.

The keywords table is a list of unique keywords, with four fields of information for each word.

keyID A keyID is a unique integer that is assigned to each keyword.

keyword This is the word itself. This field is currently limited to a length of 32 characters.

num This integer refers to the number of times the word occurs throughout the entire database.

keep This is a binary field, set to 0 by default. If it is set to 1 then that word is 'safe', and will remain on the keyword list regardless of whether or not it breaks any of the rules mentioned above.

2.2 stoplist

This is a list of words that will be removed from the keyword list, regardless of their length or how often they occur.

This table includes only one field for each unique stopword.

stopword The word itself is stored here. This field is currently limited to a length of 32 characters.

2.3 files

The files table is a list of HTML documents that have been indexed. It exists primarily so that each file can be quickly identified by an integer, but it also provides the useful function of storing information on how often a particular file is found to be the correct response for a query.

This table includes four fields for each unique file.

fileID This is a unique integer assigned to each file.

path This string stores the location of the file within the computers directory structure. This will eventually be changed to hold the URL of the document.

description This string holds the first 100 characters of the document and will be displayed along with search results to aid the CSRs in making the correct choice.

hits This field stores an integer indicating how often a particular file has been used as the correct response for a search.

2.4 searchIndex

This file describes which keywords occur in which files, how often, and how often they are used in searches. Each fileID, keyID pair indicates that the keyword represented by keyID occurs in the file represented by fileID.

The search index consists of five fields where each keyID, fileID pair is unique.

ID This stores a unique integer assigned to each keyID, fileID pair.

fileID This integer is a fileID number corresponding to an entry in the files table.

keyID This integer is a keyID number corresponding to an entry in the keywords table.

num This integer field refers to the number of times the indicated keyword occurs within the indicated file.

hits The number of time the indicated keyword has been used in a successful search for the indicated file is stored in this integer field.

2.5 queries

This table describes each query that has been used in the past. A unique query is determined by the text of the actual query, so it is possible to have multiple queries with the same keywords and the same result if the query typed by the user was different. In theory, this table will not grow to be too large because similar queries which are reused will not generate a new entry in this table. Initially the table is blank and gains entries with each unique successful search.

The queries table consists of four fields for each unique query.

queryID This is a unique integer assigned to each query.

query This string stores the actual text typed into the search engine when this query was made.

fileID This integer corresponds to an entry in the files table. It indicates the file that was chosen as the correct answer for this query.

num This integer is the number of times this query has been used to find the file.

2.6 queryIndex

This table is an extension of the queries table. It simply lists which keywords are used in each query. Each queryID, keyID pair indicates that the keyword corresponding to keyID occurs in the query that corresponds to queryID. When the engine is first used, this table is blank. It gains entries with each search that is completed.

The queryIndex consists of two fields where each queryID, keyID pair is unique.

queryID This integer is a queryID number that corresponds to an entry in the queries table.

keyID This integer is a keyID number that corresponds to an entry in the keywords table.

3 Search Algorithm Description

This section describes the procedure the search engine follows when given a query.

1. A set of keywords Ko is extracted from the query.

2. Next, a set of past queries is returned to the user, such that Ko is a subset of Kq where Kq is the set of all keywords associated with a given past query. This will return a list of queries that are similar to the query inputted by the user. The information of which queries use which keywords can be obtained by joining the queries table with the queryIndex table. The past queries that are returned will be sorted in descending order according to the number of times that query has been used.

3. If the user considers one of these past queries to be close enough to his or her own query, the record that answers the past query is displayed. If this record correctly answers the query, the appropriate counters in the various tables are incremented and the search is completed. The user can return to step 2 and try again, if the record does not correctly answer the query.

4. If none of the past queries is that which the user is looking for, then he or she is taken to the next stage of the search. A list of documents that could possibly answer the original query is returned to the user. This list will consist of every document in which Ko is a subset of Kd where Kd is the set of all keywords contained within a given document. If this list is not at least 10 items in length, then the engine will attempt to fill the list up to 10 by returning partial matches. It will return words where Kd contains every keyword in Ko but one, then every keyword but two, until a list that is ten items long is reached or until there are no more keywords to match. This list of potential documents will be sorted first by the number of keywords matched, then by a weighted average of the following factors: the number of hits the document has received, the number of times each keyword occurs within the document, and the number of times each keyword has been used to successfully 'hit' each document. For each potential document, the first 100 hundred characters of that document are displayed along with the number of keywords matched which is displayed in the form of a fraction:

$$\frac{\#matched}{\#in \quad query}$$

5. The user is also presented with a list of related keywords, if there is more than one potential document. The set of related keywords is determined by taking keywords that occur in the potential documents currently being displayed, but do not occur in the user's current query. The user can refine his or her search by adding related keywords and searching again. Keywords can also be removed in order to broaden a search.

6. Once the user finds the document that correctly answers his or her query, this new query is stored in the queries table and all of the appropriate counters throughout the database are incremented in order to increase the efficiency of the next search.

4 Current state of implementation

Currently, there are only two completed pieces of software for this search engine. This is because the bulk of the work done this past summer was put towards the tedious task of converting a Folio Infobase to HTML. Both programs are stored in the form of functions in the Getwords library. These programs are waiting to be used to make a type of Index

Manager GUI to help maintain this search engine. There is a function to build a list of keywords from a group of HTML files, and another to build a complete files table and search index from a group of HTML files. All functions listed as static in the Getwords.c file are a part of one or both of these processes. All other functions in the Getwords.c file were used to test the speed of the search engine using a DOS interface coded in C.

In the same directory as the Getwords.c file, there should be a test.c file that implements two separate main functions. The first will completely rebuild the keyword, file and searchIndex tables when given a hard-coded root directory. The second implements the DOS based search engine.

Ultimately, the code for maintaining the various indexes can be done in whatever language is favorable. All of these tasks are performed infrequently and they can be executed directly on the machine running the SQL server. Continuing to use C or C++ would be best for this code, as much of the background work has been completed.

The code for the search engine is another matter entirely. This should be done in Java so it can be incorporated directly into a frame of the web page where the documents are to be displayed. The search engine itself will consist primarily of repeating the steps of inputting data from the user, constructing the proper SQL query, and passing the query to the SQL server. If the SQL queries used in the actual Java engine are modeled after our queries used in the DOS prototype, then the search should be very efficient. Therefore, it is strongly suggested that the DOS engine be used as a starting point for building the actual engine. One should also note that the DOS prototype still performs only a "dumb" search; it skips the past query step and goes directly to a list of possible documents. Many of the principles applied to forming efficient SQL calls for the "dumb" portion of the search apply to the "intelligent" portion of the search. Adding the "intelligent" portions to the engine under the current framework should not present too many difficulties.

5 Possible improvements

- The search engine should be completed and added to the converted closed web site in such a way that it can be accessed from a web browser conveniently.
- A utility should be written to aid in the maintenance of the indexes and keyword lists. This utility should allow the administrator to perform the following functions when needed:
  - add and remove stop words (but any word that is already used as a keyword cannot be added to the stopword list once the system is in use)
  - flag keywords as "keep" and add new keywords to the current list to be flagged as "keep"
  - rebuild the keyword list and re-index the closed web site (the process currently takes about 15 minutes)

- add and remove web pages to the closed web site
- move around files within the closed web site's directory structure

- The SQL call that returns related words during a search should be altered to return relevant keywords without overwhelming the user with too much information.
- The section above describing the search algorithm refers to a weighted average of several factors involved in ranking the possible matches to a query. Someone will have to decide which factors are most important and write a formula for this average.
- There is also the potentially messy problem of dealing with the dynamic nature of the closed web site. What happens if someone adds, deletes or moves a file? What if the keyword list is re-indexed after such an addition and suddenly a keyword is pushed over one of the limits and disappears from the list? Our system is completely reliant on ID numbers, and having the item to which an ID number refers suddenly disappear or changing the item which an ID number refers to can seriously compromise the entire system. We suggest the following modifications to our model so that the engine's memory will not have to be completely wiped with each addition or alteration of the closed web site:
  - Each file's fileID should be included as a comment at the top of each file. This system is now being built into the conversion process and should be added to the add file process in the administration tool mentioned above. The fileID field of the SQL database will be determined by this comment during the indexing process, as opposed to the system currently in place which makes use of the auto-increment function of the SQL server. Therefore the file with ID #21 will remain #21 until it is removed from the database, regardless of the number of times it is moved or how many files are moved in front of it. If and when the file is removed from the closed web site, all references of past queries to #21 should also be automatically removed. This should be built into the administration tool mentioned above.
  - The problem of keywords that have been used in past searches being removed by the re-indexing process is daunting at first, but has a fairly simple solution. Before each indexing session, all words that are currently keywords must be flagged as "keep." Any keyword that may have been used in a past query cannot be deleted and their keyIDs cannot be changed. This solution makes a lot of sense: why should a word that was important in searching for a document before suddenly become unimportant because it occurs a few more times in the closed web site? Once again, this solution should be built into the administration tool.
  - Lastly, words currently on the keyword list cannot be added to the stoplist, for the same reasons listed above. This solution can be included in the administration tool.

- Because of these new limitations on the indexing process, it is very strongly suggested that any group that intends to make the most of this search engine should first use it for a one-month test period. During that period, they should compile an optimal stop list and an optimal list of "keep" words based on one month's worth of study and use. Then the entire index should be blanked, all memory of past queries and such should be erased and the group should begin fresh with the new optimal configuration. Then and only then, never to be blanked again, the program should go into full use.

- There is also the potential problem of a 24-hour call center. The current indexing process takes about 15 minutes. That may speed up slightly when the indexing is run directly on the server, but one shouldn't count on any huge improvements. That is far too long for a call center to go down for maintenance, especially if the indexing process is done once a night. In order to minimize downtime, all indexing should be done within a temporary duplicate database, and then copied over to the active tables.

- Testing should be done once the Java client is completed to make sure the system is as efficient as we have predicted. Testing should include sessions with large (20+) numbers of users querying the database at once.

6 Conclusion

The process that builds the search indexes is complete. The high-level description of the search engine itself is finished, and the existing DOS prototype of the engine works efficiently. All that remains to be done is the actual implementation of the Java search engine.

We theorize that once completely implemented, this tool should increase the efficiency of a call center. In a call center scenario, each query is answered in one unique document. Therefore, after an extended period of use, this application will become a database not only of keywords and documents, but also of questions and answers. This second format is more readable, and can therefore increase the speed at which CSRs can find the correct response to a query. It also has the added advantage of enforcing automatic sharing of information among CSRs. If one CSR finds the answer to an obscure question after a long period of searching, that question-answer pair is stored and immediately distributed to every other CSR in the center. This means that a full search only needs to be done once for any question that can be answered using the database. By allowing the search engine to learn in this manner, we are making use of information that usually goes to waste. The efficiency increase created by this application cannot be measured in terms of clock cycles or big-0 notation. Instead, one must look to the way in which it enables the human component of the search to find the document they need quickly. We firmly believe that call response time will definitely improve after this application is fully implemented and tested in an actual call center

7 Acknowledgements

The first two student authors were supported by NSERC Undergraduate Student Research Assistantship grants. The last author is supported by an NSERC Individual Research Grant.