

Ambiguity Resolution in Search Engine Using Natural Language Web Application.

Azeez Nureni Ayofe^{*1}, Azeez Raheem Ajetola¹, and Ade Stanley Oyewole²

¹Department of Maths and Computer Science, College of Natural and Applied Science,
Fountain University, Osogbo, Osun State, Nigeria.

²Computer Science Department, Federal Polytechnic Idah, P.M.B 1037 Idah, Kogi State, Nigeria.

*E-mail: nurayhn@yahoo.ca
oyewoledotbb@yahoo.com

ABSTRACT

Our aim is to create NLWA technique which will be able to retrieve resources from a knowledge base in a more efficient way to respond to the ambiguity problem that occurs when performing the search using the search engine. This system was implemented with the fundamental concept of Natural Language Processing (NLP) whereby it differentiates the similar meaning (synonyms) or multiple meaning (polysemous) of the word if it has any. The user's NL question is processed in three steps. Firstly, the linguistic pre-processing, secondly the translation of the linguistic pre-processed user question into a computer readable and unambiguous form with respect to a given ontology, and thirdly the retrieval of pertinent documents. The NLWA is an intermediary which establishes a link between it and Google search engine and able to return and generate the synonyms or differentiate the meanings of the word that input in NLWA and produces output (expected results) directly from search engine.

(Keywords: natural language, NL, NLWA, search engine, polysemous)

INTRODUCTION

The World Wide Web (WWW) can be seen as an enormous database of heterogeneous resources which is growing continuously. Query and Information retrieval is one of the central issues in WWW. We should also observe that in the absence of a formal language as SQL for databases, natural language, remains the only way for querying the web. On the other hand it is very difficult to deal with the large number of languages and the heterogeneous domains of resources. Therefore most of the Internet query tools allow as input keywords, sometimes

connected with logical operators. There are at least two consequences of this restriction: - the user who is actually concentrated on his search topic, must try to synthesize his query in this logical form, and find operators which fit to his scope.

Even with these logical operators, in the absence of a semantic representation of the query, and in parallel of the existent resources, the retrieved information will be partially out of the scope of the query. The Semantic web activities aim to give a solution to the latter point. As for the first, the only possibility to get out of the paradigms: "keywords" + "logical operators" is the use of natural language. It is however difficult to control also the complete syntax, and the level of user's language knowledge. Most part of the web users are non-native English speakers, but they are using English as query language. On the other hand, any rule-based approach in natural language analysis will make first a syntactic analysis, and even very robust (i.e. fault-tolerant) grammars fail to certain grammatical errors.

From this point of view, the empirical corpus based approach would be much more suited, but here arise again the problem of lack of data. The syntax analysis needs, when using empirical methods, tree-banks for the analyzed language. First of all, such tree banks are available for a reduced number of languages, secondly, they are not usually access free. Taking into account the above described problems, the only viable solution seems to be the use of a controlled language input, which still offers the user the power of natural language, but prevents the user from syntactic mistakes. In this paper we will present the architecture and general principles of such a system.

However, the search engines are subject to the problem of generated unwanted or irrelevant

information in the search result. The keyword(s) enter by user to perform a search may contain different meaning as represented different "sense" between verb and noun. In other words, the keywords that input may have multiple meaning which can lead to ambiguity problem. This ambiguity problem is mainly because of the search engines do not consider the exact meaning of the search query but only consider the keywords matching of the search query based on the indexes. In addition, the "keyword may not be able to convey complex search semantics a user wishes to express" and thus return the irrelevant information that are not the user desires. Due to the ambiguity problem, a prototype that applies the fundamental concept of natural language processing (NLP) was proposed.

The idea of NLWA is to differentiate the meaning behind a word and to clustering the similar meaning of a word that input to the search engine. NLWA will function on top of Google search engines as a middleware to look into the meaning of the word that enters by user and in turn performs the search in Google search engine.

Google Architecture Overview: In this section, we will give a high level overview of how the whole system works as pictured in Figure 1. Further sections will discuss the applications and data structures not mentioned in this section.

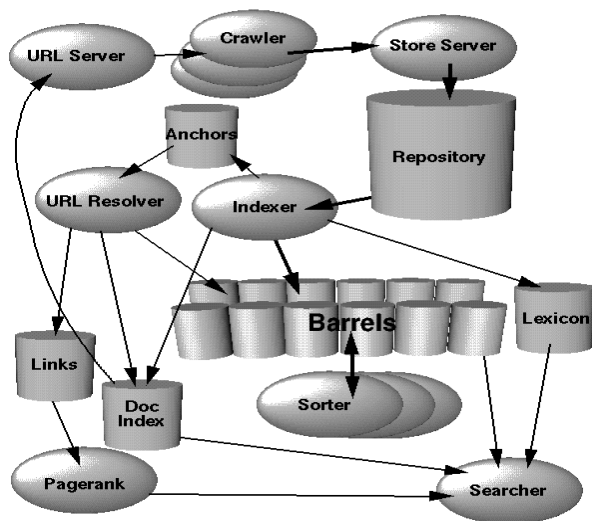


Figure 1: High Level Google Architecture.

Most of Google is implemented in C or C++ for efficiency and can run in either Solaris or Linux.

In Google, the web crawling (downloading of web pages) is done by several distributed crawlers. There is a URLserver that sends lists of URLs to be fetched to the crawlers. The web pages that are fetched are then sent to the storeserver. The storeserver then compresses and stores the web pages into a repository. Every web page has an associated ID number called a docID which is assigned whenever a new URL is parsed out of a web page. The indexing function is performed by the indexer and the sorter.

The indexer performs a number of functions. It reads the repository, un-compresses the documents, and parses them. Each document is converted into a set of word occurrences called hits. The hits record the word, position in document, an approximation of font size, and capitalization. The indexer distributes these hits into a set of "barrels", creating a partially sorted forward index. The indexer performs another important function. It parses out all the links in every web page and stores important information about them in an anchors file. This file contains enough information to determine where each link points from and to, and the text of the link.

The URLresolver reads the anchors file and converts relative URLs into absolute URLs and in turn into docIDs. It puts the anchor text into the forward index, associated with the docID that the anchor points to. It also generates a database of links which are pairs of docIDs. The links database is used to compute PageRanks for all the documents.

The sorter takes the barrels, which are sorted by docID (this is a simplification, and resorts them by wordID to generate the inverted index. This is done a list of wordIDs and offsets into the inverted index. A program called DumpLexicon takes this list together with the lexicon produced by the indexer and generates a new lexicon to be used by the searcher. The searcher is run by a web server and uses the lexicon built by DumpLexicon together with the inverted index and the PageRanks to answer queries.

Major Data Structures: Google's data structures in place so that little temporary space is needed for this operation. The sorter also produces are optimized so that a large document collection can be crawled, indexed, and searched with little cost. Although, CPUs and bulk input output rates have improved dramatically over the years, a disk seek still requires about 10 ms to complete. Google is designed to avoid disk seeks whenever possible, and this has had a considerable influence on the design of the data structures.

BigFiles: BigFiles are virtual files spanning multiple file systems and are addressable by 64 bit integers. The allocation among multiple file systems is handled automatically. The BigFiles package also handles allocation and deallocation of file descriptors, since the operating systems do not provide enough for our needs. BigFiles also support rudimentary compression options.

Repository: The repository contains the full HTML of every web page. Each page is compressed using zlib. The choice of compression technique is a tradeoff between speed and compression ratio. We chose zlib's speed over a significant improvement in compression offered by bzip. The compression rate of bzip was approximately 4 to 1 on the repository as compared to zlib's 3 to 1 compression. In the repository, the documents are stored one after the other and are prefixed by docID, length, and URL as can be seen in Figure 2. The repository requires no other data structures to be used in order to access it. This helps with data consistency and makes development much easier; we can rebuild all the other data structures from only the repository and a file which lists crawler errors.

Repository: 53.5 GB = 147.8 GB uncompressed

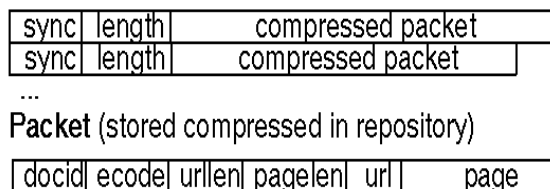


Figure 2: Repository Data Structure.

Document Index: The document index keeps information about each document. It is a fixed width ISAM (Index sequential access mode) index, ordered by docID. The information stored in each entry includes the current document status, a pointer into the repository, a document checksum, and various statistics. If the document has been crawled, it also contains a pointer into a variable width file called docinfo which contains its URL and title. Otherwise the pointer points into the URLlist which contains just the URL. This design decision was driven by the desire to have a reasonably compact data structure, and the ability to fetch a record in one disk seek during a search.

Additionally, there is a file which is used to convert URLs into docIDs. It is a list of URL checksums with their corresponding docIDs and is sorted by checksum. In order to find the docID of a particular URL, the URL's checksum is computed and a binary search is performed on the checksums file to find its docID. URLs may be converted into docIDs in batch by doing a merge with this file. This is the technique the URLresolver uses to turn URLs into docIDs. This batch mode of update is crucial because otherwise we must perform one seek for every link which assuming one disk would take more than a month for our 322 million link dataset.

Lexicon: The lexicon has several different forms. One important change from earlier systems is that the lexicon can fit in memory for a reasonable price. In the current implementation we can keep the lexicon in memory on a machine with 256 MB of main memory. The current lexicon contains 14 million words (though some rare words were not added to the lexicon). It is implemented in two parts -- a list of the words (concatenated together but separated by nulls) and a hash table of pointers. For various functions, the list of words has some auxiliary information which is beyond the scope of this paper to explain fully.

Hit Lists: A hit list corresponds to a list of occurrences of a particular word in a particular document including position, font, and capitalization information. Hit lists account for most of the space used in both the forward and the inverted indices. Because of this, it is important to represent them as efficiently as possible. We considered several alternatives for encoding position, font, and capitalization -- simple encoding (a triple of integers), a compact encoding (a hand optimized allocation of bits),

and Huffman coding. In the end we chose a hand optimized compact encoding since it required far less space than the simple encoding and far less bit manipulation than Huffman coding.

Our compact encoding uses two bytes for every hit. There are two types of hits: fancy hits and plain hits. Fancy hits include hits occurring in a URL, title, anchor text, or meta tag. Plain hits include everything else. A plain hit consists of a capitalization bit, font size, and 12 bits of word position in a document (all positions higher than 4095 are labeled 4096). Font size is represented relative to the rest of the document using three bits (only 7 values are actually used because 111 is the flag that signals a fancy hit). A fancy hit consists of a capitalization bit, the font size set to 7 to indicate it is a fancy hit, 4 bits to encode the type of fancy hit, and 8 bits of position. For anchor hits, the 8 bits of position are split into 4 bits for position in anchor and 4 bits for a hash of the docID the anchor occurs in. This gives us some limited phrase searching as long as there are not that many anchors for a particular word. We expect to update the way that anchor hits are stored to allow for greater resolution in the position and docIDhash fields. We use font size relative to the rest of the document because when searching, you do not want to rank otherwise identical documents differently just because one of the documents is in a larger font.

The length of a hit list is stored before the hits themselves. To save space, the length of the hit list is combined with the wordID in the forward index and the docID in the inverted index. This limits it to 8 and 5 bits respectively (there are some tricks which allow 8 bits to be borrowed from the wordID). If the length is longer than would fit in that many bits, an escape code is used in those bits, and the next two bytes contain the actual length.

Forward Index: The forward index is actually already partially sorted. It is stored in a number of barrels (we used 64). Each barrel holds a range of wordID's. If a document contains words that fall into a particular barrel, the docID is recorded into the barrel, followed by a list of wordID's with hitlists which correspond to those words. This scheme requires slightly more storage because of duplicated docIDs but the difference is very small for a reasonable number of buckets and saves considerable time and coding complexity in the final indexing phase done by the sorter. Furthermore, instead of storing actual wordID's,

we store each wordID as a relative difference from the minimum wordID that falls into the barrel the wordID is in. This way, we can use just 24 bits for the wordID's in the unsorted barrels, leaving 8 bits for the hit list length.

Inverted Index: The inverted index consists of the same barrels as the forward index, except that they have been processed by the sorter. For every valid wordID, the lexicon contains a pointer into the barrel that wordID falls into. It points to a doclist of docID's together with their corresponding hit lists. This doclist represents all the occurrences of that word in all documents.

An important issue is in what order the docID's should appear in the doclist. One simple solution is to store them sorted by docID. This allows for quick merging of different doclists for multiple word queries. Another option is to store them sorted by a ranking of the occurrence of the word in each document. This makes answering one word queries trivial and makes it likely that the answers to multiple word queries are near the start. However, merging is much more difficult. Also, this makes development much more difficult in that a change to the ranking function requires a rebuild of the index. We chose a compromise between these options, keeping two sets of inverted barrels -- one set for hit lists which include title or anchor hits and another set for all hit lists. This way, we check the first set of barrels first and if there are not enough matches within those barrels we check the larger ones.

Query Processor: Query processing has seven possible steps, though a system can cut these steps short and proceed to match the query to the inverted file at any of a number of places during the processing. Document processing shares many steps with query processing. More steps and more documents make the process more expensive for processing in terms of computational resources and responsiveness. However, the longer the wait for results, the higher the quality of the results. Thus, search system designers must choose what is most important to their users — time or quality. Publicly available search engines usually choose time over very high quality, having too many documents to search against.

The steps in query processing are as follows (with the option to stop processing and start matching indicated as "Matcher"):

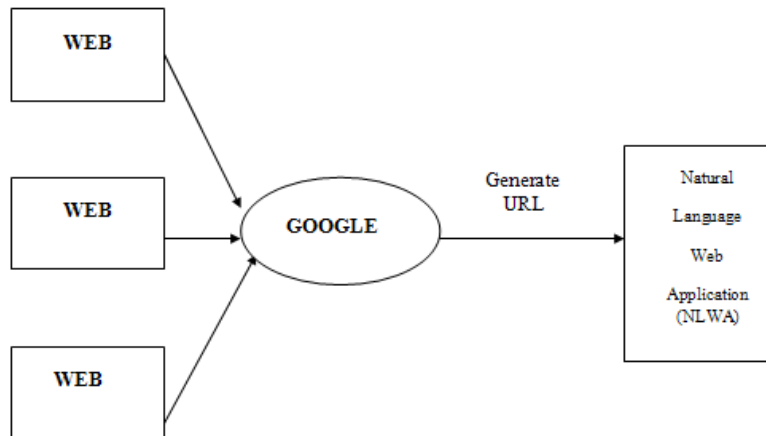


Figure 3: Concept of NLWA.

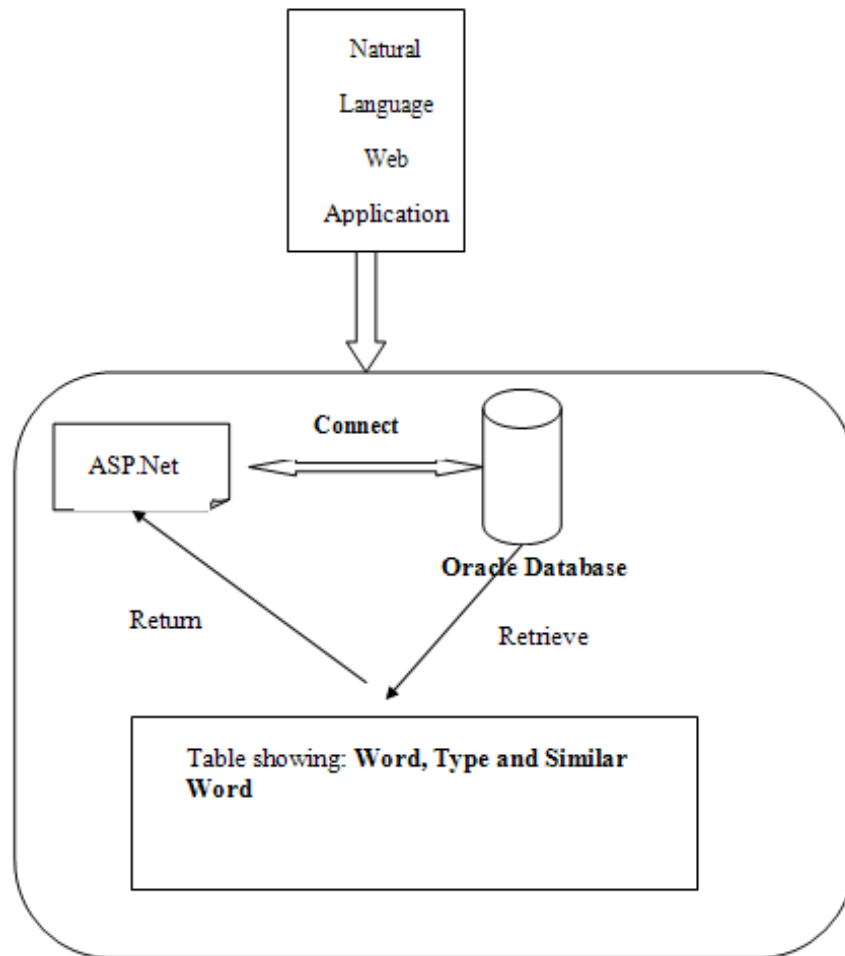


Figure 4: Framework of NLWA.

- Tokenize query terms.
Recognize query terms vs. special operators.
—————> Matcher
- Delete stop words.
- Stem words.
- Create query representation.

—————> Matcher
- Expand query terms.
- Compute weights.

—————> Matcher

Step 1: Tokenizing. As soon as a user inputs a query, the search engine — whether a keyword-based system or a full natural language processing (NLP) system — must tokenize the query stream, i.e., break it down into understandable segments. Usually a token is defined as an alpha-numeric string that occurs between white space and/or punctuation.

Step 2: Parsing. Since users may employ special operators in their query, including Boolean, adjacency, or proximity operators, the system needs to parse the query first into query terms and operators. These operators may occur in the form of reserved punctuation (e.g., quotation marks) or reserved terms in specialized format (e.g., AND, OR). In the case of an NLP system, the query processor will recognize the operators implicitly in the language used no matter how the operators might be expressed (e.g., prepositions, conjunctions, ordering).

At this point, a search engine may take the list of query terms and search them against the inverted file. In fact, this is the point at which the majority of publicly available search engines perform the search.

Steps 3 and 4: Stop list and stemming. Some search engines will go further and stop-list and stem the query, similar to the processes described above in the Document Processor section. The stop list might also contain words from commonly occurring querying phrases, such as, "I'd like information about." However, since most publicly available search engines encourage very short queries, as evidenced in the size of query window provided, the engines may drop these two steps.

Step 5: Creating the query. How each particular search engine creates a query representation depends on how the system does its matching. If a statistically based matcher is used, then the query must match the statistical representations of the documents in the system. Good statistical queries should contain many synonyms and other terms in order to create a full representation. If a Boolean matcher is utilized, then the system must create logical sets of the terms connected by AND, OR, or NOT.

An NLP system will recognize single terms, phrases, and Named Entities. If it uses any Boolean logic, it will also recognize the logical operators from Step 2 and create a representation containing logical sets of the terms to be AND'd, OR'd, or NOT'd.

At this point, a search engine may take the query representation and perform the search against the inverted file. More advanced search engines may take two further steps.

Step 6: Query expansion. Since users of search engines usually include only a single statement of their information needs in a query, it becomes highly probable that the information they need may be expressed using synonyms, rather than the exact query terms, in the documents which the search engine searches against. Therefore, more sophisticated systems may expand the query into all possible synonymous terms and perhaps even broader and narrower terms.

This process approaches what search intermediaries did for end users in the earlier days of commercial search systems. Back then, intermediaries might have used the same controlled vocabulary or thesaurus used by the indexers who assigned subject descriptors to documents. Today, resources such as WordNet are generally available, or specialized expansion facilities may take the initial query and enlarge it by adding associated vocabulary.

Step 7: Query term weighting (assuming more than one query term). The final step in query processing involves computing weights for the terms in the query. Sometimes the user controls this step by indicating either how much to weight each term or simply which term or concept in the query matters most and *must* appear in each retrieved document to ensure relevance.

Leaving the weighting up to the user is not common, because research has shown that users are not particularly good at determining the relative importance of terms in their queries. They can't make this determination for several reasons. First, they don't know what else exists in the database, and document terms are weighted by being compared to the database as a whole. Second, most users seek information about an unfamiliar subject, so they may not know the correct terminology.

Few search engines implement system-based query weighting, but some do an implicit weighting by treating the first term(s) in a query as having higher significance. The engines use this information to provide a list of documents/pages to the user.

After this final step, the expanded, weighted query is searched against the inverted file of documents.

OPERATION OF NATURAL LANGUAGE FOR WEB APPLICATION (NLWA)

NLWA work as the middle engine, try to cluster the similar meaning of word and in return on the search in search engines by differentiates between the types of word. The differentiation is done by looking into the type of word respective in verb, noun, and adjective to obtain the meaning behind the word. The purpose of differentiate is to disambiguate the word that has multiple meaning.

This is because certain word has multiple meaning in different type either is verb or noun which present different of meaning may lead to the returned of irrelevant search results. As the meaning that present for the word that is polysemous are normally totally different. Additionally, the purpose of clustering is to group the similar meaning or synonymy of the word together in order to increase the chances of getting the information that user desire by looking into the meaning and the type of the particular word.

Internal Process of NLWA: The figure above shows the specific view of the flow of internal process from the high level view system framework. The ASP.Net will connect to the database and retrieve the dictionary data from the table that stored in the database.

After retrieved the data, it then return the value in order to generate the URL. Specifically, during the retrieving process, the ASP.Net will read row by row to look for the data that stored in the table "WORD" column. In this case, when it matches the data that stored in the "WORD" column based on the word that user key in, it will return the value that stored in the "SIMILAR_MEANING" column to ASP.Net.

The value refers to the similar meaning of word which is clustered in the database according to the meaning of the particular word. Lastly, when the value returns to ASP.Net, it will pass in each returned value as parameter to generate with the URL together in order to link to Google. With the parameters that pass it, Google is being informed what to search.

APPLICATION OF NLWA

A test for some selected words is applied so as to evaluate the actual meaning behind the words. NLWA will differentiate the types and meaning of "choose" for example and produces the equivalent meaning. **Natural language processing (NLP)** has the advantages of break the impasse and open up the possibilities of the Semantic Web. First, NLP systems can now automatically create annotations from unstructured text. This provides the data that semantic web applications require.

Second, NLP systems are themselves consumers of semantic web information and thus provide economic motivation for people to create and maintain such information. A new generation of natural language search systems can take advantage of semantic web markup and ontologies to augment their interpretation of underlying textual content. They can also expose semantic web services directly in response to natural language queries.

Instead of clustering word that has similar meaning, NLWA differentiated words that has multiple meaning as well. Again, the input of a word "choose" to the NLWA, the differentiation between the types and meaning of the word "choose" is processed. Thus, the input word has multiple meanings of "order" as shown in the figure below.

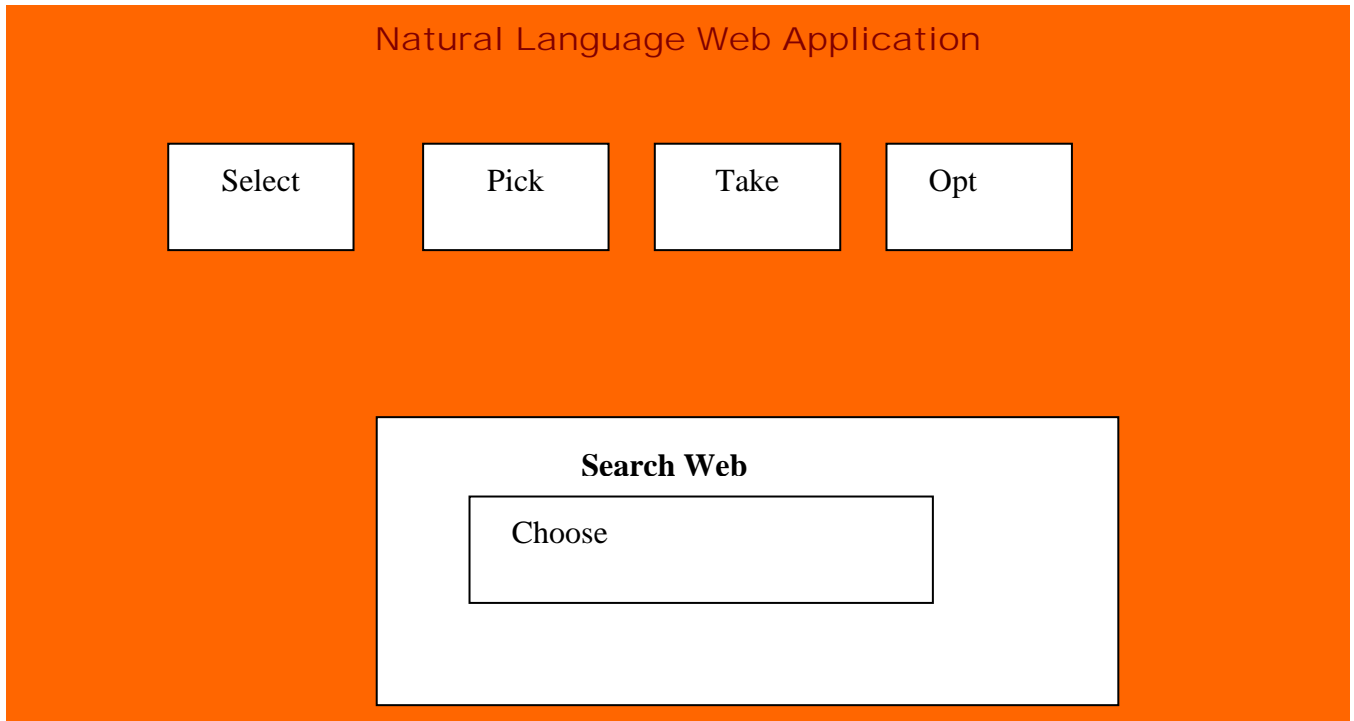


Figure 5: Testing page on input of keyword “choose” in NLWA.

However, Google search engine returned the search result according to the parameter on the meaning that a user might select. By showing all the meanings of the word that is polysemous, users can review which sense of meaning that the word present and select which they want to search.

As a result, in comparing the results gathered from NLWA to the normal search on Google’s, researchers find that the results returned applying the concept of NLWA has improved and also solved the ambiguity problem that occurs in performing the search using the search engine.

CONCLUSIONS

This paper has presented the how search engines work and explored the architecture of each. Secondly, based on the research hypothesis that has been proved, the ambiguity words may lead to the return of irrelevant search result of the search engines. Moreover, based on the ambiguity problem of a word that is polysemous, the solution is designed to identify the meaning of word or synonyms for the word that is input in the

Natural Language Web Application (NLWA) are working. With the dictionary data which clustering for the similar meaning of word that stored in the database, the prototype is able to return either the meaning or synonyms of the word that input based on the type of the word in verb, noun, or adjective. Also, the test shows that the NLWA consists of the function that described and able to return the appropriate parameters.

REFERENCES

1. Allen, J. 1994. *Natural Language Understanding*. Addison Wesley, New York, NY.
2. Katz, B. 1997. “Annotating the World Wide Web using Natural Language”. 5th RIAO Conference on Computer Assisted Information Searching on the Internet. Montreal, Canada.
3. Bridge, D. 2004. “Natural Language Processing (NLP)”. University College Cork, Ireland, www.cs.ucc.ie/dgb/courses/ai/notes/notes41.pdf.
4. Qiang Yang, Hai-Feng Wang, Ji-Rong Wen, Gao Zhang, Ye Lu¹, Kai-Fu Lee, and Hong-Jiang Zhang. 2000. “Towards A Next-Generation Search Engine”, 1.

5. TechWeb Network. 2008. "Search Engines", <http://www.techweb.com/encyclopedia/defineterm.jhtml?term=Search+Engine>. 24 February 2008.
6. Gracia, J., R. Trillo, M. Espinoza, and E. Mena, 2006. "Querying the Web: A Multontology Disambiguation Method". ACM Press, University of Zaragoza, 22 February 2008. 241-242.
7. Sullivan, D. 2007. "How Search Engines Work". Search Engine Watch, <http://searchenginewatch.com/showPage.html?page=216803> 18 March 2008.
8. Shapiro, Y. and E. Lehoczky. 2003. "How Do Search Engines Work?". SearchEngines.com. http://www.searchengines.com/search_engines_101.html. 25 February 2008.
9. Liddy, E. 2001. "How a Search Engine Works ". Director of the Center for Natural Language Processing Professor, School of Information Studies, Syracuse University, Vol. 9 No. 5, <http://www.infotoday.com/searcher/may01/liddy.htm>. 26 February 2008.
10. UC Berkeley Library. 2008. "How do search engines work?". <http://www.lib.berkeley.edu/TeachingLib/Guides/Internet/SearchEngines.html>. 25 February 2008.
11. Franklin, C. 2008. "How Internet Search Engines Work". <http://www.howstuffworks.com/search-engine.htm>. 26 February 2008.
12. Boswell, W. 2008. "How Do Search Engines Work?". <http://websearch.about.com/od/enginesanddirectories/>

SUGGESTED CITATION

Ayofe, A.N., A.R. Ajetola and A.S. Oyewole. 2009. "Ambiguity Resolution in Search Engine Using Natural Language Web Application". *Pacific Journal of Science and Technology*. 10(2):453-461.

