LIS4031: Project #1

Jose Carlos Archundia Adriano - Carlos Andres Reyes Evangelista
jose.archundiaao@udlap.mx - carlos.reyesea@udlap.mx

Universidad de las Américas Puebla — September 6, 2019

Introduction

This report aims to illustrate the implementation of the Boolean Retrieval Model, as well as the features that a retrieval model like this one might have, including its advantages, disadvantages and its inherent relevance.

The program that will be covered in this report implements a boolean retrieval model using a posting list, and was developped using *Lua*, a lightweight but powerful scripting language with a built-in support for pattern matching. This feature is not as powerful as a regular expression matcher, but effective enough to perform most of the operations required to accomplish the goal natively.

1 Boolean Retrieval Model

1.1 Theoretical viewpoint

This model is one of the first retrieval models created and it is currently still being used. According to Bruce Croft et. al. (2009), this contemporary relevance is mainly due to its ease of implementation and its efficiency when it comes to the elimination of non-relevant documents during the scoring process. Nevertheless, there are several downsides to take into account when considering using it. "The effectiveness depends entirely on the user. Because of the lack of a sophisticated algorithm, simple queries will not work well", says Croft when referring to its major drawback.

The behavior of the boolean retrieval model is straightforward: the queries may contain terms and boolean operators (which may be associated) and the set of retrieved documents consists *only* of documents whose content fulfills these boolean requirements. This model works under the assumption that all retrieved documents are equivalent in relevance as long as they contain the terms indicated in the query. The latter statement is the reason why this retrieval model is not considered a ranking algorithm.

1.2 Algorithmic issues

One of the problems with the boolean retrieval model is the fact that it uses an incidence matrix. This means that the frequency of the terms are not taken into account. In the worst case, the algorithm might be returning documents that only have one incidence of a term instead of a document with many more. To overcome this problem, a posting list, with a frequency for each term and a list of documents in which the term appears, is used. Another problem is the fact that the program needs to parse user input for the queries to be searched. This means that user input must be validated before the query is processed. On top of this, in order to process the query, the order in which to execute the operators of the query has to be descided by the program. This step requires extra processing of the user query to accomplish.

2 Implementation

Document parsing

In order to generate the posting list, we first needed to extract the relevant information from the document list. To do this, we cut the document list into individual document strings by matching from one ".I" tag to another. After identifying individual documents, we then created an object in memory for each

one, with fields for each one of the tags *I,T,A,B,W*. By giving the data a defined structure in memory, we were then able to easily access the desired information of any one of the documents.

Posting list creation

To create the posting list, we took advantage of the fact that in *Lua* you can create data structures that contain key and value pairs. Each token that we found in the documents became a key in this data structure, and the value became a linked list of all the documents that contained that token. This means that we are able to rapidly able to create the posting list, and also able to rapidly access the document list for any given token.

Query validation and parsing

One of the problems that the program faces when parseing the query is the validation of the user's input. We want to inform the user of an error in the query when the program is unable to understandt it. At the same time however, the program tries to *correct* the query by ignoring misplaced tokens and only uses the ones that may be used to form a valid query.

This algorithm basically splits the query into a head and tail. The head contains the first term of the query along with its parentheses and **not** operator, if they exist. The tail is the rest of the query, and is processed in *pairs*. These *pairs* only match parts of the query that contain an **and** or **or** operator followed valid combinations of spaces, parentheses, **not** operators, and an alphanumeric term. This implies that only the **valid** parts of the query are considered for future processing.

Consecuently, if the user enters a query with a incorrect syntax but valid data, the system will only consider the part that it can process. For instance, the query $standard\&\|acess$ will be interpreted as $standard\|access$.

Query execution

Evaluating an expression with operators that have a specific precedence is not a simple task, since the program is required to evaluate certain parts of the expression first, independently of where are they found. To address this issue, the query was transformed from an infix-form (more human-readable) to a postfix-form (more machine-readable).

This procedure is not complicated; using the **valid** expression generated above, the steps are as follows: A stack is declared to temporarily store the operators and a list that stores the pre-constructed postfix form. Each infix token is then processed one by one. If it is a term, it is immediately pushed into the postfix list. If it is an opening parenthesis, it is immediately pushed into the operators stack. If it is any other operator, then its precedence is compared to the one on the top of the stack. Operators with a greater precedence are pushed, otherwise, all the operators precedence greater than or equal to [?] are popped from the stack and pushed into the list. Finally, if the processing token is a closing parenthesis, then all the operators in the stack are popped until the next opening parenthesis is found. If an opening parenthesis is found when all the infix tokens are processed, or if a closing parenthesis is found before an opening one, then an exception is returned.

Once the postfix form is obtained, it is processed by pushing all the terms into a new stack. Every time an operator is found, the required amount of parameters are popped from the stack and used to perform that operation, and will push the result into the stack. The remaining term in the stack is the final result of the expression.

3 Conclusions

Throughout the development of a program that implements the Boolean Retrieval model using a posting list, we were able to come to understand the nuances of using this approach. We were able to experience firsthand the strengths and weaknesses of this model. Our implementation of the Boolean Retrieval Model allowed us to deepen our understanding of it, and helped to solidify our knowledge on the topic.

4 References

Croft, W., Metzler, D. and Strohman, T. (2009). *Information retrieval in practice.* Upper Saddle River, N.J.: Pearson Education.