Using Structured Queries for Source Code Search

Brian P. Eddy
The University of Alabama
Tuscaloosa, AL, USA
bpeddy@ua.edu

Nicholas A. Kraft
ABB Corporate Research
Raleigh, NC, USA
nicholas.a.kraft@us.abb.com

Abstract—Software maintenance tasks such as feature location and traceability link recovery are search-oriented. Most of the recently proposed approaches for automation of search-oriented tasks are based on a traditional text retrieval (TR) model in which documents are unstructured representations of text and queries consist only of keywords. Because source code has structure, approaches based on a structured retrieval model may yield improved performance. Indeed, Saha et al. recently proposed a feature location technique based on structured retrieval that offers improved performance relative to a technique based on traditional TR. Although they use abstract syntax tree (AST) information to structure documents, they nonetheless use content-only (keyword) queries to retrieve documents. In this paper we propose an approach to source code search using AST information to structure queries in addition to documents. Such queries, known as content and structure (CAS) queries, allow developers to search for source code entities based not only on content relevance, but also on structural similarity. After introducing the structured retrieval model, we provide examples that illustrate the trade-off between the simplicity of content-only queries and the power of CAS queries.

Keywords—Software maintenance; program comprehension; text retrieval; structured document retrieval; static analysis

I. INTRODUCTION

Much recent work on source code search is based on the application of text retrieval (TR) models such as latent semantic indexing (LSI) and latent Dirichlet allocation (LDA). In such work, TR models are applied to corpora in which each document contains the text associated with a source code entity such as a class or method. Despite source code being structured (by the syntax of the language in which it is written), nearly all work on source code search represents source code documents as bags of words — unstructured sets of terms.

Documents extracted from source code can be structured using abstract syntax tree (AST) information. Consider a document representing a method. Distinct sections can represent the signature and body. Within the signature, fields can represent the method name, and a subsection can represent the parameter list, within which fields can represent the parameter types and names. Similarly, sections and fields can represent parts of the method body, including statements and their constituent expressions. Note that a given term may appear in only the signature (e.g., a parameter type) or in both the signature and body (e.g., a parameter name). Unlike a traditional TR model, in which the structure of a method is not considered, a structured retrieval model allows a developer to issue a query that includes terms and sections/fields of interest.

Context is key in source code search. Identifiers play many roles in source code, and an identifier can convey different information in different contexts. For example, suppose the identifier test appears in the name of one method and in a method call in the body of another method. In the former case, test may relate to the main responsibility of the method, whereas in the latter case, test may relate to one step in a larger responsibility. Using content and structure to search a corpus and retrieving either the most relevant section(s) or document(s) based on content relevance and structural similarity is known as structured document retrieval (SDR) [1]. Unlike traditional TR models, SDR models support powerful query languages in which a user may specify several constraints, including the scope of the query and the weight or probability assigned to each term or structural entity.

In this paper we contribute:

- Methodology for source code search using content and structure (CAS) queries
- Motivating examples for and a preliminary study of the use of CAS queries for source code search

II. BACKGROUND AND RELATED WORK

In this section we review background and related work.

A. Related Work

Saha et al. [2] propose the use of structured retrieval for software search. They introduce BLUiR, a bug localization tool based on the Indri\(^1\) search engine. Saha et al. extract structured documents from source code by traversing the Eclipse AST. They extract four fields of text (class names, method names, variables, and comments) and sum the results of eight distinct queries to calculate the relevancy score for each document. Using 3,400 bugs, Saha et al. demonstrate that BLUiR can improve on the accuracy of techniques based on traditional TR models. However, BLUiR uses little of the available AST information, and they use content-only queries rather than the more powerful content-and-structure queries that are possible via the Indri Query Language.

Bassett and Kraft [3] propose the use of structural term weighting to improve the accuracy of source code search. Unlike previous work on incorporating structural information in the search process (e.g., [4], [5]), which augment the search

\(^1\)http://www.lemurproject.org/indri
by fusing its results with static or dynamic analysis results, Bassett and Kraft incorporate structural information in the TR model via term weighting. They study the effects of different term weighting schemes in which the importance of method names and method calls are adjusted relative to other terms in a method document. Using over 400 bugs, Bassett and Kraft demonstrate that increasing the importance of method names tends to improve the accuracy of a TR-based source code search for their subject systems, but that increasing the importance of method calls tends to degrade accuracy. In Section V we describe how structural term weighting can be achieved using CAS queries.

B. Language Models

In our preliminary study (see Section V) we use the Indri search engine for TR. Indri is based on language modeling [6], a statistical technique that models the distribution of terms and term phrases in a document. In a language model, the relevance of a document to a query is the predicted likelihood that the document produces the terms in the query. Assuming a unigram model (meaning that the model represents individual terms but not term phrases), the probability of a query being generated by the language model is:

$$p(Q|M_d) = \prod_{t \in Q} p(t|M_d) \times \prod_{t \notin Q} 1 - p(t|M_d)$$

The first term in the equation is the probability of generating the query terms, and the second term in the equation is the probability of not generating additional terms. The simplest way to calculate the probability of a term being generated by a language model is to divide the raw term frequency by the total number of terms in a document. However, there are two key problems with this simple approach. First, a document only reflects part of a language model. Most language models handle this problem by finding the mean probability of a term across all documents in which the term is present, mixing it with the probability of the document. Because each document could come from a different language model, a risk factor is used to determine how much to emphasize the mean probability. The updated equation, with risk factor $R_{i,d}$, is:

$$\hat{p}(Q|M_d) = p_{na}(t,d)^{(1-R_{i,d})} \times p_{avg}(t)^{R_{i,d}}$$

The second problem with the simple approach arises when $p(t|M_d) = 0$ (i.e., when a term does not occur in a document). In this case, using the simplest calculation for the probability of a query would yield $p(Q|M_d) = 0$. For this reason, smoothing algorithms are used to predict the probability of terms not present in a given document. An example of a simple smoothing algorithm would be to use the ratio of all occurrences of a term in the corpus to the total number of terms in the corpus. For any document in which the term is not present, this ratio is used instead of 0. Other, more sophisticated, smoothing algorithms may be designed depending on the corpus.

Language modeling can be combined with SDR [7]. In such approaches, each part of the document is represented either by its own language model, or by the set of child language models for its subparts. Returning to the example from Section I, each method is represented by its own language model, which is interpolated from the language models of the method’s signature and body (and so on).

III. Approach

In this section we describe our approach, including the Indri retrieval model on which it is based.

A. The Indri Retrieval Model

Before Indri can index a corpus and build a model, the user needs to provide the search engine with a parameters file. This parameters file contains both fields and metadata about the corpus being indexed. For the purposes of our research, we make direct use of the fields that can be specified in the parameters. Fields are extents of the textual content of a document (e.g. a heading or body tag in an HTML document). Indexed field names are available for use in an Indri query language query. In a query, the user may specify the fields they wish to search. By specifying fields and using an appropriate file format, users may develop their own structured formats for indexing in the Indri system.

The Indri Retrieval Model combines inference networks for information retrieval [8] with language models discussed in Section II. An inference network is a method of defining a joint probability distribution over a collection of random variables. It is represented as a directed acyclic graph where each node in the graph corresponds to random variables. Edges in an inference network represent conditional dependencies where nodes that have no connection are independent. With each node is an associated probability function that takes as input the variable from the node’s parent variables and gives as output the random variable represented by the node. Documents and language models are represented as nodes in the inference network.

B. Creating a Structured Corpus

We have developed a tool to extract structured documents from methods in source code. The tool takes as input a set of source code files written in the Java programming language and parses the files using ANTLR² (the tool is extensible to other languages or to make the current output of the tool more expressive for different documents, all that is needed is a new grammar). Tokens are extracted from the document and tagged with their source entity (e.g., method name, comment). The tool maintains order of tokens and folds leading comments into their respective methods by associating comments immediately preceding a method with that method. These tokens are then fed to a preprocessor that performs splitting, case normalization, stop word removal, and stemming. When performing splitting both the split and original tokens are maintained, however this can be configured to change. The resulting terms are then grouped into documents and placed into a corpus. The next step is the document formatter that reads the corpus and is given as input a structure for the new method documents. The formatter then outputs a corpus of structured documents in the specified format and an Indri parameters file.

We discuss document structures further in Section IV.

²http://www.antlr.org/
C. Creating Structured Queries

Current queries used in feature location take the general approach of utilizing the description or title from a bug or issue report as a query. This approach is due to the need in research to make the most general queries possible without making assumptions about the developer. While these queries fit the purposes of such studies, very little research has focused on using developer knowledge about a system to improve the results of a query. For instance, a developer may have an understanding of what terms are actually used in the system, what terms relate to method names and class names, and what terms refer to variables or fields. A developer might also have expectations of what context a term is used in. Allowing a more robust query system that allows developer input can increase the likelihood of returning relevant results. Furthermore, such a query system would be complementary to existing query refinement and reformulation techniques and provide additional input to such a technique about a term’s context. We wish to leverage a developer’s knowledge during the software search process. For this reason, a proper query language should allow for multiple options and be highly configurable. The Indri query language allows for a wide variety of queries. The simplest queries in Indri take the form:

```
#combine("side" "computes" Point)
```

The # signifies a query, combine means to search for the terms together in a document, while the query is provided in the parentheses. For such a query, each term in the query is given equal weighting and the query is issued across all fields. This query does not make use of the structural document, but instead uses the document as a bag of words where each term is given an equal weight. The quotation marks around a term indicate a search for the unstemmed/unnormalized version of the term. Lack of quotations searches for the stemmed/normalized terms. These queries represent content only queries. These queries try to match over all terms in a query instead of based on fields and each field is of equal importance. Developer knowledge is not incorporated into this phase of the retrieval process.

Queries are not limited to these basic types however. It is possible to search for queries based on fields:

```
#combine[signature](Point areaTriangle)
#combine[method_name](areaTriangle)
#combine[signature](#combine[.method_name](area triangle)) #combine[/parameter_name](point1))
```

In these queries, the fields appear in the square brackets while the query appears in the parentheses. Each of these queries makes use of the field to increase the likelihood that a particular method will be found. The examples above all search for particular terms in a method signature. The first of these queries looks for the terms over all terms in the signature (i.e., the method name, parameter type, and parameter name). The second query is more specific and searches directly for all methods with the name “areaTriangle.” This query is useful if the developer already knows the terms to be a part of the name of methods in the program. The final query is a nested query. It searches for terms in the method signature but also searches for “areaTriangle” in the method name and “point1” in the terms for parameters. Each of these queries returns increasingly specific results and is based on what the developer knows about the source code.

The nested query above is of particular interest. We will talk more about how this is made possible in Section IV.

In addition to the basic and structural queries, Indri allows for terms and queries to be weighted. Weighting queries opens up another avenue for increasing accuracy. In the case of topic modeling, increasing the weight of terms from method names and decreasing the weight of terms from method calls was shown to increase search results. If the developer has certain beliefs about the usage or location of terms, they may choose to vary the weights of terms in queries or the weights of nested queries. The basic structure of a weighted query is:

```
#weight( 2.0 #combine[signature](area) 1.0
#combine[body](area))
```

In the example query, the developer has given greater weight to documents with area appearing in the method signature versus the method body. Perhaps, the developer knows the method they want computes an area, so they believe that area is likely to be in the method name. They want to see methods with area in the method name before other possible choices. By weighting the query, they are more likely to retrieve the results they want. While developers may manually weight their queries, we also wish to identify queries that may return higher results for developers that are less familiar with the system or suggest queries that may produce more relevant results to the developer. There are multiple ways to suggest queries in this case. Two possible approaches include issuing the same query across each field in a source document but varying the weight of each field according to some heuristic or suggesting fields in which a term is most likely to be found. Future research is required to refine these approaches.

IV. STRUCTURED CORPUS

In this section we discuss the creation of a structured corpus and the implications of different types of structured method documents. While the objective is to provide the flexibility needed by the developer, there are important considerations to take into account during creation.

The traditional approach to text retrieval techniques on software search is to treat method documents as flat files of text with no structure. Indri is able to perform this operation using an inference network with a single language model node. However, terms in source code may be categorized due to implicit structural information. For example, a developer reading code is able to recognize variables, methods, classes, literals, class fields, comments, etc., due to the usage and relationship of the terms to the code. Furthermore, a developer is able to recognize whether a term is a part of a declaration or use. This information can be extracted automatically by tools to build structured documents from source code.

The structure in a source code document is implicit and is understood in different ways. Therefore it is possible to create different structural representations of a method document. As an example, terms in source code can be broken down into three broad categories: identifiers, literals, and comments.
/** Computes the area of a triangle using Heron's formula. */

public static double areaTriangle(
    Point point1, Point point2, Point point3) {
    // compute the lengths of the sides
    double side1 = computeDistance(point1, point2);
    double side2 = computeDistance(point2, point3);
    double side3 = computeDistance(point3, point1);
    // compute half of the perimeter
    double p = (side1 + side2 + side3) / 2;
    return Math.sqrt(p * (p - side2) * (p - side3));
}

Fig. 1. Example Method

Fig. 2. Example Document

Each of these three categories reflect the different ways that terms can be used. Identifiers for instance define the state and behaviors of the system. Literals are terms that are found in input and output operations. Terms in comments are important to the developer as they give additional information that is not present in the program’s instructions. For each structural component in the document, a new model node is created. An inference network with these three categories would require language model nodes for each of the three categories. While this type of document is simple, it allows for quick and easy searches over all of a method’s identifiers, comments, and literals. This type of model for the structured document reduces the complexity for querying. However, it is not as robust or as flexible as it could be.

As another approach, Figure 1 gives an example of a Java method that may be present in the source code of a system. Inspecting this method, we can identify several categories for terms. The first line of the method is a leading comment related to the method. In the method signature we can identify the method name, parameter types, and parameter names. In the body of the method, we can identify method calls, local variables, line comments, and references. An example of a structured document with this information is given in Figure 2 using XML formatting. The root of the document is divided into three main categories, the method comment, the method signature, and the body. Method comments precede methods. Method comments are believed to be used for defining the responsibilities of a method and therefore provide important terms for searching. The method signature can be broken down further into method name, parameter names, and parameter types. We use a nesting structure to allow for multiple types of queries at different levels of granularity. The goal is to allow for more flexible queries that will lead to an increase in accuracy. Similarly, we nest terms appearing in the body.

This document allows for increased flexibility in the search queries. Any of the structural components can now be queried and with the proper parameters file, indexed by the search engine. However, while this improves the flexibility for certain tasks, it decreases the flexibility for others. For instance, if the developer wishes to query the system over all variables, the query needs to be explicit on which fields to search. This applies to comments and literals as well. Furthermore, it is possible that a finer granularity may not lead to any improvement in relevant results obtained from a structured query. This is due to two reasons. First, there needs to be additional study on what types of queries and granularities the developer is most likely to use. Second, with each new structural component we add, we increase the number of language models for a document and these language models increasingly smaller in size. Language models are an approximation of the usage of terms in a particular context. Therefore, the more information available for the language model, the better that approximation can be. Defining too many components increases the overall complexity of the system while the benefits for doing so should start to see diminishing returns.

Furthermore, the complexity continues to increase not only in the modeling of components, but also in the network. Every field specified in a parameters file given to Indri becomes indexable, meaning that every field creates a new sub document node under the original method document.

The choice of structure used for a structured method document should be based on the system being indexed, what is needed to complete the tasks, and the resources available for the process.

<table>
<thead>
<tr>
<th>TABLE I. EDIT RESULTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight(S,B)</td>
</tr>
<tr>
<td>(1,1)</td>
</tr>
<tr>
<td>(1,2)</td>
</tr>
<tr>
<td>(1,4)</td>
</tr>
<tr>
<td>(1,6)</td>
</tr>
<tr>
<td>(2,1)</td>
</tr>
<tr>
<td>(4,1)</td>
</tr>
<tr>
<td>(8,1)</td>
</tr>
</tbody>
</table>

V. STRUCTURED QUERIES

In this section we discuss the effects of weighting terms according to their structural component. The objective is to show how structural weighting may be used to decrease or increase the rankings of relevant documents in the software search task. Understanding the effects of different weighting schemes is important when making recommendations of queries to new developers.
In muCommander, all three features show improvement as the weighting on the Body component increases. There may be multiple reasons for this improvement in the ranking as the weighting on the body component increases. The first is that the low number of bugs selected for this experiment, the importance on the method body may not be typical. With a larger number of features, it may be shown that the method signature has better performance overall when weighted higher. The importance of this study is not to make that distinction. This experiment is focused on demonstrating that a change in the structural weighting of method documents has an effect on the ranking of relevant methods.

The second possible cause is the simplicity of the weighting method. In this experiment we chose to break a method document into the method signature and the method body. These two components contain multiple subcomponents that are not fully considered in the result of these queries. For instance, the method signature contains the method name, the parameter type, and the parameter name. Similarly, the method body contains local variables, comments, string literals, method calls, and other references and types. Therefore the weighting scheme used in this experiment is weighting these two composite groups.

There are other options. If in the method signature, parameter names and types lower the performance of the software search while the method name improves results, then with the proper structure and queries, Indri can be made to weight the method names higher in the final result. An area of future research is in determining the best weighting schemes for a given software system.

VI. CONCLUSIONS AND FUTURE WORK

In this paper we proposed a new methodology for searching source code, discussed ways to structure a source code document, and showed how to formulate CAS queries for source code. More work is needed to evaluate and improve our methodology. Research directions include:

- Identification of the best general weighting scheme for particular software systems
- Improving recommendations for new developers by mining historical data of experienced developers
- Optimizing smoothing parameters for use with source code
- Combining existing query methodologies with structured queries on source code

REFERENCES


