Sourcerer: A Search Engine for Open Source Code

Sushil Bajracharya*, Trung Ngo*, Erik Linstead†, Paul Rigor†, Yimeng Dou†, Pierre Baldi†, Cristina Lopes*
*Institute for Software Research †Institute for Genomics and Bioinformatics
Donald Bren School of Information and Computer Sciences
University of California, Irvine
{sbajrach,trungcn,elinstea,prigor,ydou,pfbaldi,lopes}@ics.uci.edu

Abstract

`sourcerer` *is a search engine for open source code that extracts fine-grained structural information from the code. This information is used both to implement a basic notion of* code rank *and to enable search forms that go beyond conventional keyword-based searches.* `sourcerer` *supports two types of searches: (1) implementations, and their use; and (2) program structures.*

Several schemes were compared for ranking the results of code search. Results are reported involving 1,555 open source Java projects, corresponding to 254 thousand classes and 17 million LOCs. Of the schemes compared, the scheme that produced the best search results was one consisting of a combination of (a) the standard TF-IDF technique over Fully Qualified Names (FQNs) of code entities, with (b) a "boosting" factor for terms found towards the right-most handside of FQNs, and (c) a composition with a graph-rank algorithm that identifies popular classes.

1 Introduction

With the popularity of the Open Source Software movement [32], there has been an increasingly wide availability of high quality source code over the Internet. This often makes developers view the Internet as a *Scrapheap* for collecting the raw materials they use in production in form of some reusable component, library, or simply examples showing implementation details [1].

Quantity carries with it the problem of finding relevant and trusted information. A few years ago, there were only a few open source projects, and developers knew what those were, what they did, and where they were on the Internet. These days, the scenario is quite different. For example Sourceforge hosts more than 128,000 open source projects, Freshmeat has a record of more than 41,000 projects and Tigris has more than 900 high quality open source projects. When going to well-known repositories, it is difficult to find relevant components, much less relevant pieces of code. Moreover, the available projects are of varying quality. Some of these sites provide simple searching capabilities, which are essentially keyword-based searches over the projects' meta-data.

This paper focuses on the problem of source code search over Open Source projects available on the Internet. When looking for source code on the Internet, developers usually resort to powerful general-purpose search engines, such as Google. Web search engines perform well for keyword-based search of unstructured information, but they are unaware of the specificities of software, so the relevant results are usually hard to find. Recently there has been some initiative in developing large scale source-code-specific search engines [2, 6, 5, 3]. While these systems are promising, they do not seem to leverage the various complex relations present in the code, and therefore have limited features and search performance.

In [35], code search is reported as the most common activity for software engineers. Sim et al. summarize a good list of motivations for code search [33] where the use of several search forms – such as looking for implementations of functions and looking for all places that call a function – stand out. Unfortunately, these empirical studies are too small, slightly outdated, and made in the context of proprietary code. Even though several forms of code search have been heavily used in software engineering tools, and, recently, in code search engines, the concept of *code search* still has plenty of room for innovative interpretations. What motivations, goals, and strategies do developers have when they search for open source code?

Without empirical data to answer that question, but based on Sim's study and on informal surveys, we use the following basic goals for `sourcerer`: (a) *Search for implementations:* A user is looking for the implementation of certain functionality. This can be at different levels of granularity: a whole component, an algorithm, or a function. (b) *Search for uses:* A user is looking for uses of an existing piece of code. Again, this can be at different levels of granularity: a

whole component, a collection of functions as a whole, or a function. (c) *Search for characteristic structural properties:* A user is looking for code with certain properties or patterns. For example, one might be looking for code that includes concurrency constructs.

A practical and useful application of structure-based search is the identification of benchmark code for hardware testing. Benchmark selection is essentially concerned with code complexity, focusing on the generation of test cases with particular branching, looping, and concurrency properties. By allowing these properties to be specified in parameterized queries the effort required to collect a comprehensive suite of benchmark software is greatly minimized.

We have developed an infrastructure to extract and store features from source code found in repositories on the web, that can serve the search goals mentioned above. The first part of this paper describes the core components of our infrastructure, and the search application, called `sourcerer`.

Finding source code that matches some search criteria is only the first step of a search engine. The second, and more difficult, step is to present the (possibly thousands of) results using some measure of *relevancy*, so that the time that people spend searching for relevant code is minimized. Code search on Open Source projects being a topic of increasing interest, there are several proposals, prototypes, and, recently, products that have the same goal of `sourcerer`. The approaches that they use to the issue of ranking the results vary from conventional text-based ranking methods, to graph-based methods. Using the `sourcerer` infrastructure, we performed a comparative study of four schemes for ranking source code; the results are reported here. The system, and all supporting materials for this paper, are available at http://sourcerer.ics.uci.edu/about.html.

The rest of the paper is organized as follows. Section 2 presents the architecture of `sourcerer`. Section 3 describes the source code feature extraction and storage in detail. Section 4 presents the heuristics we studied for ranking search results and Section 5 presents our assessment methodology and assesses the performance of those ranking methods. In Section 6 we present related work. Finally Section 7 concludes the paper.

2 Architecture

Figure 1 shows the architecture of our search engine, `sourcerer`. The arrows show the flow of information between various components. Information on each system component is given below.

- *External Code Repositories:* These are the source code repositories available on the Internet.
- *Code Crawlers and Automated Downloads:* We have several kinds of crawlers: some target well-known repositories, such as Sourceforge, others act as web spiders that look for arbitrary code available from web servers. An automated dependency and version management component provides managed download of these repositories.
- *Local Code Repository:* The system maintains a local copy of each significant release of the projects, as well as project specific meta-data. The scheme conforms to the project object model as given by the Maven build system (`http://maven.apache.org`).
- *Code Database:* This is the relational database that stores *features* of the source code. Part of this information is described in Section 3. We are using PostgresSQL 8.0.1.
- *Parser / Feature Extractor:* A specialized parser parses every source file from a project in the local repository and extracts entities, fingerprints, keywords and relations. These *features* are extracted in multiple passes and stored in the relational database. Some highlights of this component are described in Section 3.
- *Text Search Engine (Lucene):* The keywords coming out from the parser, along with information about related entities, are fed into a text search engine. We are using Lucene 1.9.1 (`http://lucene.apache.org`).
- *Ranker:* The ranker performs additional non-text ranking of entities. The relations table from the code database is used to compute ranks for the entities using ranking techniques as discussed in Section 4.

A custom task scheduler runs multiple instances of the final three components (Parser+Lucene+Ranker) for parallel indexing of multiple repositories. Currently a Sunfire x4100 machine with 2.6Ghz Dual Core AMD Opteron Processor and 16GB RAM is used for indexing. A latest instance of indexing 146 unique versions of selected projects amounting to more than 6 million SLOCs took about 14 hours on this machine. While this is quite acceptable for a prototype that validates the search application, we intend to improve the performance when we scale this up to substantially more projects (we're planning to scale up to 100K).

With these components, we have a general purpose infrastructure for indexing source code. The search application was built using the indexed keys and ranked entities. As shown in Figure 1 (Sample UI) a list of keywords entered by a user are matched against the set of keywords maintained by the Lucene. Each key is mapped to a list of entities. Each entity has its rank associated with it. Each entity also

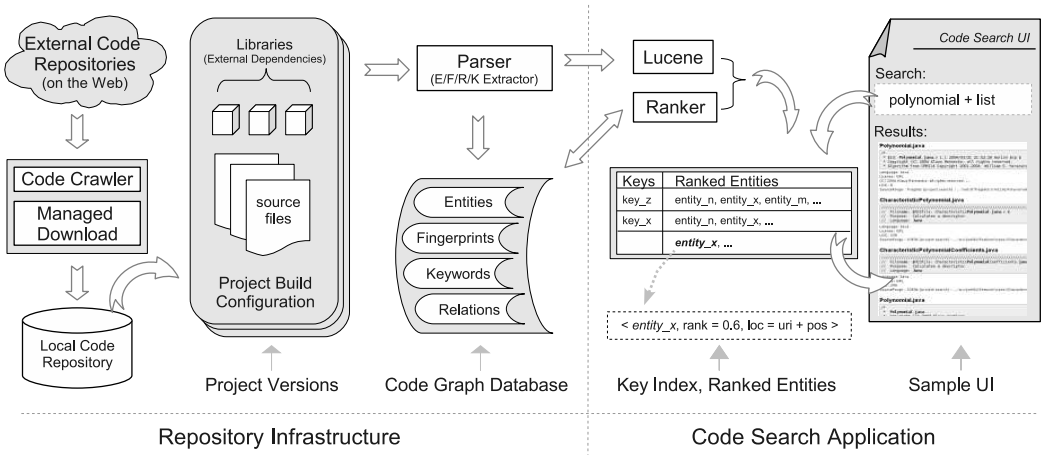

Figure 1. Architecture of the Sourcerer infrastructure

has other associated information such as its location in the source code, version, and location in the local repository to fetch the source file from. Thus, the list of entities that are matched against the sets of matching keys can be presented to the user as search results with all this information attached in a convenient way.

3 Feature Extraction and Storage

Implementing a conventional search engine (or tool) involves extracting pertinent *features* from the artifacts to be searched, and storing them in an efficiently indexed form. In the case of `sourcerer`, we are exploring several heuristics for ranking search results that involve complex structural information that can't be obtained simply with string matching. This section describes the most important structures and processes associated with the extraction and storage of source code features in `sourcerer`.

3.1 Relational Representation

Various storage models have been used to represent source code and each have their limitations and benefits [13]. Given the scale of our system, we base it on a relational model of source code.[1]

One direct benefit of using a relational model is that it provides structural information explicitly laid out in a database. However, breaking down the model into finer level of granularity can make querying inefficient, both in expression and execution [13]. Therefore, the model and design of our database schema (program entities and their relations) were carefully assessed. Rather than having tables for each type of program entity, which would be a relational representation of the Abstract Syntax Tree that would be inefficient for querying, we use a source model consisting of only of two tables: (i) program entities, and (ii) their relations. Additionally, we also store compact representations of attributes for fast retrieval of search results. Two such representations are a) indexed keywords, and b) *fingerprints* of the entities. All these entities, relations, and attributes are extracted from the source code using a specialized parser that works in multiple passes. We discuss these elements of our model for Java.

1. *Entities:* Entities are uniquely identifiable elements from the source code. Declarations produce unique program entity records in the database. Program entities can be identified with a fully qualified name (FQN) or they can be anonymous. The list of entities that are extracted during the parsing steps are: (i) package, (ii) interface, (iii) class (including inner class), (iv) method, (v) constructor, (vi) field, and (vii) initializer.

   When the parser encounters any of these declarations it records an entry in the database assigning the following attributes to the entity: a FQN, document location in the local repository that associates version and location information (where declared) with the entity, position and length of the entity in the original source file, and a set of meta-data as name-value pairs. These include a set of keywords extracted from the FQN. The meta-data field is extensible as needed.

2. *Relations:* Any dependency between two entities is represented as a relation. A dependency d originating from a source entity s to a target entity t is stored as a relation r from s to t. The list of relations are: (1) *inside*, implying lexical containment, (2) *extends*, (3) *implements*, (4) *calls*, (5) *throws*, (6) *returns*, (7)

[1] Relational models have been widely used for representing source code. See [23, 11, 10, 17], among others.

instantiates, (8) *reads*, (9) *writes* , (10) *overrides* and (11) *overloads*. Besides these, a more generic relation *uses* is used to capture other dependencies that are not captured using the above.

3. *Keywords:* Keywords are meaningful text associated with entities; they are more appropriate for humans than for machines. The parser extracts keywords from two different sources: FQNs and comments. Keyword extraction from FQNs is based on language-specific heuristics that follows the commonly practiced naming conventions in that language. For example, the Java class name "QuickSort" will generate the keywords "Quick" and "Sort". Keywords extraction from comments is done by parsing the text for natural and meaningful keywords. These keywords are then mapped to the entities that have unique IDs and that are close to the comment, for example the immediate entity that follows the comment. These keywords are also associated with the immediate parent entity inside which the comment appears. The current version of `sourcerer` does not fully exploit the specificity and meta-information in Java comments. We will be adding this feature soon.

4. *Fingerprints:* Fingerprints are quantifiable features associated with entities. Because of their importance, they are described next.

3.2 Fingerprints

A fingerprint is a sequence of numbers, implemented as a d-dimensional vector, that describes quantifiable features of some entity. This allows similarity to be measured using techniques such as cosine distance and inner product computation, well-known techniques in information retrieval [9].

For purposes of open source code search, we define the fingerprint of a code entity as a compact representation of its properties, some of them structural in nature, such as the presence of particular concurrency construct, others heuristic in nature, such as maintainability metrics [27, 37]. Fingerprints are created for each code entity as the source is parsed, and stored along with other information in the database.

Fingerprints are used in `sourcerer` to support structural searches of source code. A structure-based query needs only to refer to entity fingerprints to find relevant hits, and can thus avoid unnecessarily complicated SQL statement generation and the corresponding processing. We have defined three types of fingerprints:

- *Control structure fingerprint.* This captures the control structure of an entity. It currently contains 13 attributes including the number of synchronization statements, the number of loops, and the number of conditionals. It is associated primarily with method entities, and it is then cumulatively associated with class and package entities.

- *Type fingerprint.* This captures information about a type. It currently contains 17 attributes including the number of implemented interfaces, the number of declared methods, and the number of fields. It is associated primarily with class entities, and it is then cumulatively associated with package entities.

- *Micro patterns fingerprint.* This captures information about implementation patterns, also known as micro patterns [16]. It contains 23 selected attributes from the catalog of 27 micro patterns presented in [16]. Some examples are the "designator" (an interface with no members), the "stateless" (a class with no fields other than static final ones), and the "extender" (a class which extends the inherited protocol, without overriding any methods).

As stated before, our fingerprints were designed to support a variety of structure-based searches efficiently. Explicit structural searches are important for developers of software-related projects who are looking for benchmark or test code. But fingerprints are also the basis for supporting similarity measures among different pieces of code. So, for example, `sourcerer` includes a feature that consists in finding similar code to the code returned as a search result.

3.3 Inter-Project Links

Unlike in a regular compiler, we are looking for *source* code, and not just interface information. For example, when parsing some class A, the parser may encounter a reference to a class named B that is not part of the project's source code. Maybe the project being parsed includes the bytecodes version of B, or maybe not. If the bytecodes exist, then the parser can resolve the name B and find its interface. If the bytecodes don't exist, then this can't be done. In any case, the source-to-source link from A to B cannot be directly made, in general, during parsing of A – for example, the project where B is declared may not have been parsed yet or may even not be part of the local repository at the moment.

Our system resolves source-to-source inter-project links in an additional pass through the database. This process is not as simple as it seems, because we have designed `sourcerer` to be able to cope with multiple versions of projects. As such, we have devised a method that establishes source-to-source inter-project links by finding the most likely version of the target project that the source

project may be using. This method is relatively complex to be explained here, and is not relevant for the purposes of this paper, therefore we skip its explanation. The main point to be made is that `sourcerer` is able to resolve source-to-source inter-project links very accurately, even in the presence of multiple versions of the projects. These links are critical for the graph-based algorithms that we are using and for additional ones that we will be using in the future.

4 Ranking Search Entities

Finding information is only the first step of a search engine. In the case of source code, a method as simple as 'grep' will yield results. The second, and more difficult, step is to present the information using some measure of *relevancy* with respect to the terms being searched. We present the ranking methods that we have implemented and that are assessed in the next section.

Baseline: Code-as-Text

One of the most well-known and successful methods for ranking generic text documents is term frequency - inverted document frequency (TF-IDF). TF-IDF's heuristic is that the relevancy of a document with respect to a search term is a function of the number of occurrences of the search term in that document (TF) multiplied by the number of documents in which the term occurs (IDF). This method is at the heart of Lucene, a well-known text retrieval tool that is available as open source. We fed the source code files, as text, to Lucene, and used that as the baseline ranking method.[2]

As free text goes, TF-IDF is very effective. We know, however, that source code is not free text: it follows a strict structure and several well-known conventions that can be used to the advantage of ranking methods.

Heuristic 1: Packages, Classes, and Methods Only

When searching for implementations of functionality we can assume that those pieces of functionality are wrapped either in classes or in methods, and aren't simply anonymous blocks of code. Moreover, most likely the developers have named those code entities with meaningful names. This knowledge suggests focusing the search on names of packages, classes, and methods names, and ignoring everything else. We did this as a first heuristic, and then applied TF-IDF to this much smaller collection of words.

Heuristic 2: Specificity

We also know that there is a strict containment relation between packages, classes, and methods, in this order, and that developers use this to organize their thoughts/designs. So a hit on a package name is less valuable than a hit on a class name, which, in turn, is less valuable than a hit on a method name.

Heuristic 3: Popularity

In their essence, programs are best modeled as graphs with labels. As such, it is worth exploring other possible ranking methods that work over graphs. In Inoue et al. (Spars-J) [21], it has been suggested that a variation of Google's PageRank [22] is also an effective technique for source code search. In their Component Rank model, a collection of software components is represented by a weighted directed graph, where the nodes correspond to components (classes) and the edges correspond to cross component usage. A graph rank algorithm is then applied to compute component ranks based on analyzing actual usage relations of the components and propagating the significance through the usage relations. Using the component rank model, classes frequently used by other classes (i.e. *popular* classes) have higher ranks than nonstandard and special classes.

We took Spars-J's Component Rank method and reimplemented it in `sourcerer` as *Code Rank*, different from Component Rank in the following ways:

- Our method is more general and fine-grained than Spars-J's in that we apply it not only to classes but also to fields, methods and packages. Spars-J's method works well when the user is interested in searching the most relevant classes. However, if the query term resides in methods instead of classes, it's unclear how the the method will perform.
- Our method is less sophisticated that Spars-J's in that it doesn't perform pre-processing of the graph to cluster similar code. This simplification is justified for the purpose of the assessment goals (see Section 5).
- We exclude all classes from the JDK, because they rank the highest but developers know them well; as such, they are not that interesting.

Additional comparisons between Spars-J and our work are given in Section 6.

We used Google's PageRank almost verbatim. The Code Rank of a code entity (package, class, or method) A is given by: $PR(A) = (1-d) + d(PR(T_1)/C(T_1) + ... + PR(T_n)/C(T_n))$ where $T_1...T_n$ are the code entities referring to A, $C(A)$ is the number of outgoing links of A, and d is a damping factor.

5 Assessing `sourcerer`

Open source code search engines can be assessed using the standard information retrieval metrics *recall* and *precision*. *Recall* is the ratio of the number of relevant records

[2] Note that what we did here is nothing but a glorified 'grep' on source files. As baseline it is quite appropriate, because (1) Lucene is a freely available tool providing powerful text retrieval facilities, and (2) anyone can easily replicate this baseline.

retrieved to the total number of relevant records available; *precision* is the ratio of the number of relevant records retrieved to the total number of irrelevant and relevant records retrieved. For the purposes of this paper, we fixed the *precision* parameter, and we focused mainly on *recall* issues. Specifically, we focused on recall of the records that fall within the top 10 and 20 ranking positions.

Comparisons between `sourcerer` and other search engines are problematic, because there is no benchmark dataset. Therefore, we skip such comparisons. We note that other work on source code search engines has included comparisons with Google (see e.g. [21]). While that is interesting, it's not controlled enough to address the assessment goals we are pursuing. Our assessment methodology is comparable to that used in [25].

5.1 Assessment Goal

Ultimately, what one wants of an open source code engine is that it presents the most relevant search hits on the first page of results. Therefore, the question for which we want to find an answer is: what is the best combination of heuristics for ranking the results of open source code search? The results presented here are far from answering this question, but the methodology and conclusions from this study point us in the right direction. The goal of this assessment was to compare the text-based, structure-based and graph-based methods described in Section 4 and combinations of those methods.

5.2 Methodology

For achieving our assessment goal we used carefully chosen control queries and studied the results for those queries. Control queries have the following properties: (1) they result in a reasonable number of hits, large enough for diversity of results but small enough to be manually analyzable by people; and (2) their search intent is obvious, so that it is easy to agree which hits are the most relevant.

Once control queries are selected, a person (an Oracle) analyzes the resulting hits and decides on the N best hits for each query, where N is a number between 3 and 10. The following criteria were used to select "good hits" from the result set in the most systematic way possible: (1) Content corresponding to the search intent; (2) Quality of the result in terms of completeness of the solution; (3) The "reputation" of the project from which the solution originates.

We defined 10 control queries (see Table 1) to be run against an indexed code repository containing 1555 open source Java projects, corresponding to 254 thousand classes and 17 million SLOCs. The queries themselves were chosen to represent the various intentions of searchers as much as possible. To this end, queries were formulated that would be both indicative of a casual user, perhaps searching for an implementation of a standard data structure (eg. a bounded buffer), as well as a more advanced user interested in implementations on a larger scale (eg. a complete ftp server).

Code	*Query*	*Best hits*
Q1	bounded buffer	5
Q2	quick sort	10
Q3	depth first search	5
Q4	regular expression	3
Q5	tic tac toe	3
Q6	ftp server	5
Q7	tcp server	10
Q8	rmi server	5
Q9	chat server	7
Q9	ftp client	4
	Total best hits:	*57*

Table 1. Control queries used in the assessment of `sourcerer`.

The queries were formulated so that they matched the desired intent given the indexing methods. For the baseline measurements, code-as-text with Lucene, query (*term*) was formulated as (*term**), query (*term*$_1$ *term*$_2$) was formulated as (*term*$_1$*term*$_2$ *OR (term*$_1$* *AND term*$_2$**)), etc. For all the other measurements, and given that term unfolding is done during parsing and indexing according to some basic Java naming guidelines, query (*term*) was simply (*term*), query (*term*$_1$ *term*$_2$) was (*term*$_1$ *AND term*$_2$), etc. The two formulations don't fully match, in the sense that they don't retrieve the exact same records; but they are close enough for the baseline to have the same total *recall* as the others: in the baseline measurements, only 5 out of 57 best hits were not retrieved.

Finally, the different ranking schemes are compared with respect to the positions at which they place these N best hits for each control query. A perfect ranking scheme would place the N best hits for every query on the N top-most positions of the result set.

5.3 Results

Figure 2 shows a snapshot of the results for four ranking schemes and the baseline. The plot displays the search results for each query, highlighting the positions at which the best hits rank. The difference in search performance of the ranking methods is visible upon informal observation: the improvements correspond to having more best hits at the top. The differences are particularly noticeable from the third set of results (FQNs + coderank) to the forth (FQNs + right-hand side boost), with the last set (all methods com-

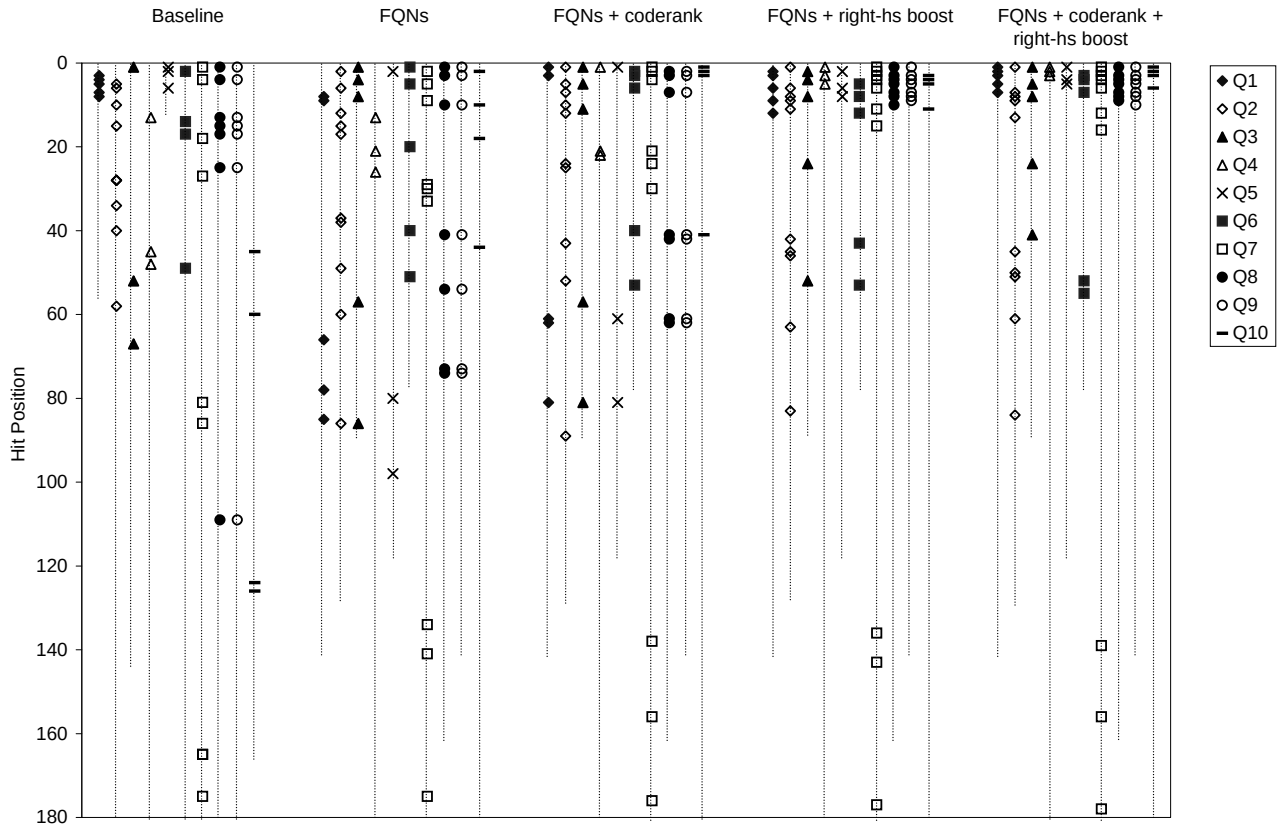

Figure 2. Placement of best hits by the different ranking schemes.

Ranking scheme	*Top 10*	*Top 20*
Baseline (code-as-text)	30%	44%
FQNs	31%	42%
FQNs + coderank	40%	44%
FQNs + right-hs boost	63%	74%
FQNs + coderank + right-hs boost	67%	74%

Table 2. Recall of best hits within top positions – top 10 and top 20 positions shown.

bined) standing out as the best. These visual observations are captured numerically in Table 2.

Given this data, we can draw the following conclusions:

- *Comparison between indexing all code as text (baseline) and looking only at names of code entities (FQNs only).* The performance of these two schemes is very similar. What this means is that, of all the text in source files, the names of the entities are the main source of effective information about functionality, and everything else is noise.
- *Effect of Code Rank (FQNs + coderank).* Taking into account the popularity of code entities, according to the sheer number of uses that they have, shows a small improvement over looking at FQNs only. Since we are counting all the references, not just inter-project ones, we believe that the improvement is due to issues of size: (1) larger projects tend to have classes with higher Code Rank, because they have more intra-project references; and (2) larger projects usually have good code; hence the improvement.
- *Effect of specificity (FQNs + right-hs boost).* This heuristic is the one that has the strongest effect on the results. Taking into account the hierarchical decomposition of the designs enables the search method to place more relevant entities in the top positions of the result set.

6 Related Work

Our work builds on a large body of work from different parts of Computer Science and Software Engineering.

Search Engines

General purpose search engines, such as Google, are "live", in the sense that they have the ability to constantly find information without the sources of that information having to report or register it explicitly, or even know that they are being analyzed. That is one of the goals of `sourcerer`, and, as such, it diverges from traditional software management tools that operate over well-identified sets of source code that are collected together through some external protocol. Google's PageRank [22] was also the inspiration behind our code rank algorithm, as it was for other work before ours. It is known that Google evolved from that simple heuristic to include dozens of additional heuristics pertaining to the structure of the web and of web documents. But general-purpose search engines are unaware of the specificities of source code, treating those files as any other text file on the web. Searching for source code in Google, for example, is not easy, and Google, by itself, is incapable of supporting the source-code-specific search features that we are developing.

Recent commercial work is bringing the capabilities of web search engines into code search. Koders [2], Krugle [6], Codase [5], and csourcesearch [3] are the few prominent ones. While developing open source code search engines can, indeed, be done these days with off-the-shelf software components and large amounts of disk space, the quality of search results is a critical issue for which there are no well-known solutions. While researching these systems, we found a variety of user interfaces to search and explore open source code, but we were very often disappointed with the search results themselves. Those systems being proprietary, we were unable to make comparisons between their ranking methods and ours. A bottom-line comparison of the results also didn't produce meaningful conclusions, because the datasets that those products operate on are dramatically different from the dataset we used for our study.

Two projects have recently been described in the research literature that use variants of the PageRank technique to rank code, namely Spars-J and GRIDLE. Spars-J [7, 20, 21] is the project closest to ours. Their Component Rank technique is a more sophisticated variation of our code rank technique, in that it performs pre-processing on the graph in order to cluster classes with similar (copy-and-paste) code. The reported results of that work confirmed that it is possible to detect some notion of relevancy of a component (class) based solely on analyzing the dependency graph of those components. However, when analyzing that work we found important questions that were left unanswered. Specifically, it was unclear how the improvements shown by this graph-based heuristic compared to other, simpler, heuristics for source code search. That was, essentially, the question that we tried to answer in our work so far. According to the results we got, it seems that something as simple as focusing the search on class and method names, and defining appropriate rules for weighting the parts of those names, produces very good results, albeit using a different concept of relevancy. Finally, it seems that combining these two heuristics improves the results even further. The main contribution of our work over Spars-J is, therefore, the realization that improvements on the quality of search results require not just one, but a combination of several heuristics, some graph-based, others not.

GRIDLE [31] is a search engine designed to find highly relevant classes from a repository. The search results (classes) are ranked using a variant of PageRank algorithm on the graph of class usage links. GRIDLE parses Javadoc documentation instead of the source code to build the class graph. Thus it ignores more fine grained entities and relations.

Software Engineering Tools

Modern software engineering tools are bringing more sophisticated search capabilities into development environments extending their traditionally limited browsing and searching capabilities [19, 24, 34, 30]. Of particular interest to this paper are the various ranking techniques and the search space these tools use.

Prospector uses a simple heuristic to rank *jungloids* (code fragments) by length [24]. Given a pair of classes (T_{in}, T_{out}) it searches the program graph and presents to the user a ranked list of jungloids, each of which can produce class T_{out} given a class T_{in}. Its ranking heuristic is conceptually elegant but too simple to rank the relevance of search results in a large repository of programs where every search need not be for getting T_{out} given T_{in}.

Stratchoma uses structural context from the code user is working on and automatically formulates a query to retrieve code samples with similar context from a repository [19]. A combination of several heuristics is used to retrieve the samples. The results are ranked based on the highest number of structural relations contained in the results. The search purpose Stratchoma fulfills is only one of the many possible scenarios possible. Nevertheless, the heuristics implemented in it are good candidates to employ in searching for usage of framework classes.

JSearch [34] and JIRiss [30] are two other tools that employ Information Retrieval (IR) techniques to code search. JSearch indexes source code using Lucene after extracting interesting syntactic entities whereas JIRiss uses Latent Semantic Indexing [25]. Both these tools lack graph based techniques and thus are limited to their specific IR specific

ranking in presenting results.

Fingerprints

The fingerprint technique is used successfully across domains as a means for comparing entities for structural similarity. A notable instance has been in the chemistry domain, where this approach is used to encode and search molecular data [15, 8]. In the computing domain, fingerprints are used for security [26].

We found very few examples of using code fingerprints specially for the purposes of search but that may be because the software community uses alternative terms for denoting the same concept. Hill and Rideout use method fingerprints for automatic method completion [18]. A similar idea is employed in the Software Bookshelf project [14], for representing software entities in terms of metrics. Our fingerprints also have similar goals to work focusing on finding source code patterns [28, 29]. Our fingerprints are less expressive, in the sense that they don't support full pattern matching; but they are very efficient for search, as they involve only simple numerical computations. They are meant to capture quantifiable attributes of entities, and to support extremely fast search.

7 Conclusions

We have presented `sourcerer`, a search engine for open source code search built on an infrastructure that extracts fine-grained structural information about source code. In the process of developing `sourcerer`, we have been experimenting with several heuristics, as well as with assessment methodologies that can tell us whether those heuristics are useful or not. In this paper, we described 3 such heuristics and showed their performance using our assessment methodology. The heuristic that showed the highest improvement in relevancy of search results was one that includes a combination of text-based, structure-based, and graph-based ranking methods. We believe that this will be true for future work in devising new heuristics for open source code search: no single heuristic, not even a single class of heuristics, will be enough.

So far we have been focusing on improving the search for implementations of functionality, but we are well aware that this is not the only need that developers have with respect to using open source code. We are working on leveraging the rich structural information that is stored on the database in order to provide new features such as uses of classes and methods, how to use frameworks, change impact analysis, and determining the smallest set of dependencies for a given source file. While none of these features is new, we believe their impact will be much greater if they work on the whole collection of open source projects on the Internet, without the developers having to follow additional protocols for identification and indexing of projects, which is what happens with current IDEs. As such our work is complementary to the work in IDEs, and there is a good synergy between the two. We are currently working on integrating these search capabilities `sourcerer` with Eclipse using a Web Service that exports the search facilities as APIs.

A more general conclusion drawn from this work is the realization that if open source code engines are to be developed in a friendly competitive manner, the community needs to come together to establish a benchmark against which the different methods can be compared.

References

[1] Scrapheap Challenge Workshop, OOPSLA 2005. *http://www.postmodernprogramming.org/scrapheap/ workshop*.

[2] Koders web site. *http://www.koders.com*.

[3] csourcesearch web site. *http://csourcesearch.net/*.

[4] Codecrawler web site. *http://codecrawler.sourceforge.net/*.

[5] Codase web site. *http://www.Codase.com*.

[6] Krugle web site. *http://www.krugle.com*.

[7] SparsJ Search System. *http://demo.spars.info/*.

[8] Daylight Theory Manual. *http://www.daylight.com/ dayhtml/doc/theory/theory.toc.html*.

[9] R. A. Baeza-Yates and B. A. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press / Addison-Wesley, 1999.

[10] R. I. Bull, A. Trevors, A. J. Malton, and M. W. Godfrey. Semantic grep: Regular expressions + relational abstraction. In *WC'02 9th Working Conference on Reverse Engineering*, pages 267– 276, 2002.

[11] Y.-F. Chen, M. Y. Nishimoto, and C. V. Ramamoorthy. The c information abstraction system. *IEEE Trans. Softw. Eng.*, 16(3):325–334, 1990.

[12] R. Conradi and B. Westfechtel. Version models for software configuration management. *ACM Comput. Surv.*, 30(2):232–282, 1998.

[13] A. Cox, C. Clarke, and S. Sim. A model independent source code repository. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 1. IBM Press, 1999.

[14] P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Mueller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong. The software bookshelf. *IBM Systems Journal*, 36(4):564–593, Nov. 1997.

[15] M. Fligner, J. Verducci, and P. Blower. A modification of the jaccard/tanimoto similarity index for diverse selection of chemical compounds using binary strings. In *Technometrics*, volume 44, pages 1–10, 2002.

[16] J. Y. Gil and I. Maman. Micro patterns in java code. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 97–116, New York, NY, USA, 2005. ACM Press.

[17] E. Hajiyev, M. Verbaere, and O. de Moor. Codequest: Scalable source code queries with datalog. In *Proceedings of the 2006 European Conference on Object-Oriented Programming (to appear)*, July 2006.

[18] R. Hill and J. Rideout. Automatic method completion. In *ASE*, pages 228–235. IEEE Computer Society, 2004.

[19] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 117–125, New York, NY, USA, 2005. ACM Press.

[20] K. Inoue, R. Yokomori, H. Fujiwara, T. Yamamoto, M. Matsushita, and S. Kusumoto. Component rank: relative significance rank for software component search. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 14–24, Washington, DC, USA, 2003. IEEE Computer Society.

[21] K. Inoue, R. Yokomori, T. Yamamoto, and S. Kusumoto. Ranking significance of software components based on use relations. *IEEE Transactions on Software Engineering*, 31(3):213–225, 2005.

[22] R. M. Lawrence Page, Sergey Brin and T. Winograd. The pagerank citation ranking: Bringing order to the web. *Stanford Digital Library working paper SIDL-WP-1999-0120 of 11/11/1999 (see: http://dbpubs.stanford.edu/pub/1999-66)*.

[23] M. A. Linton. Implementing relational views of programs. In *SDE 1: Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 132–140, New York, NY, USA, 1984. ACM Press.

[24] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the api jungle. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 48–61, New York, NY, USA, 2005. ACM Press.

[25] A. Marcus, A. Sergeyev, V. Rajlich, and J. Maletic. An information retrieval approach to concept location in source code. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE 2004)*, pages 214–223, Nov. 2004.

[26] NIST. Secure hash standard. http://www.itl.nist.gov/fipspubs/fip180-1.htm, 1995.

[27] P. Oman and J. Hagemeister. Metrics for assessing a software system's maintainability. In *Proceedings of the International Conference on Software Maintenance 1992*, pages 337–344. IEEE Computer Society Press, Nov. 1992.

[28] S. Paul. Scruple: a reengineer's tool for source code search. In *CASCON '92: Proceedings of the 1992 conference of the Centre for Advanced Studies on Collaborative research*, pages 329–346. IBM Press, 1992.

[29] S. Paul and A. Prakash. A framework for source code search using program patterns. *IEEE Trans. Softw. Eng.*, 20(6):463–475, 1994.

[30] D. Poshyvanyk, A. Marcus, and Y. Dong. Jiriss - an eclipse plug-in for source code exploration. *icpc*, 0:252–255, 2006.

[31] D. Puppin and F. Silvestri. The social network of java classes. In H. Haddad, editor, *SAC*, pages 1409–1413. ACM, 2006.

[32] E. S. Raymond. The catheral and the bazaar. *http://www.catb.org/ esr/writings/cathedral-bazaar/cathedral-bazaar/*.

[33] S. E. Sim, C. L. A. Clarke, and R. C. Holt. Archetypal source code searches: A survey of software developers and maintainers. In *IWPC*, page 180, 1998.

[34] R. Sindhgatta. Using an information retrieval system to retrieve source code samples. In L. J. Osterweil, H. D. Rombach, and M. L. Soffa, editors, *ICSE*, pages 905–908. ACM, 2006.

[35] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil. An examination of software engineering work practices. In *CASCON '97: Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*, page 21. IBM Press, 1997.

[36] D. Spinellis and C. Szyperski. How is open source affecting software development? *IEEE Software*, 21(1), Jan.-Feb. 2004.

[37] K. D. Welker and P. W. Oman. Software maintainability metrics models in practice. In *Crosstalk, Journal of Defense Software Engineering*, volume 8, pages 19–23, November/December 1995.