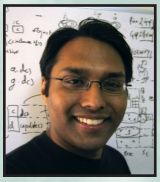
ISR Institute for Software Research
University of California, Irvine

A Study of Ranking Schemes in Internet-Scale Code Search



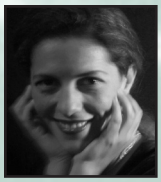
Sushil Bajracharya
University of California, Irvine
sbajrach@ics.uci.edu

Yimeng Dou
University of California, Irvine
ydou@ics.uci.edu

Trung Ngo
University of California, Irvine
trungcn@ics.uci.edu

Pierre Baldi
University of California, Irvine
pfbaldi@ics.uci.edu

Erik Linstead
University of California, Irvine
elinstea@ics.uci.edu



Cristina Lopes
University of California, Irvine
lopes@ics.uci.edu

Paul Rigor
University of California, Irvine
prigor@ics.uci.edu

November 2007

ISR Technical Report # UCI-ISR-07-8

Institute for Software Research
ICS2 217
University of California, Irvine
Irvine, CA 92697-3455
www.isr.uci.edu

A Study of Ranking Schemes in Internet-Scale Code Search

Sushil Bajracharya*, Trung Ngo*, Erik Linstead†, Paul Rigor†, Yimeng Dou†,
Pierre Baldi†, Cristina Lopes*
*Institute for Software Research †Institute for Genomics and Bioinformatics
{sbajrach,trungcn,elinstea,prigor,ydou,pfbaldi,lopes}@ics.uci.edu

Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3423
ISR Technical Report # UCI-ISR-07-08
November 2007

Abstract. The large availability of source code on the Internet is enabling the emergence of specialized search engines that retrieve source code in response to a query. The ability to perform search at this scale amplifies some of the problems that also exist when search is performed at single-project level. Specifically, the number of hits can be several orders of magnitude higher, and the variety of conventions much broader.

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 results using some measure of relevance with respect to the terms being searched.

We present an assessment of 4 heuristics for ranking code search results. This assessment was performed using Sourcerer, a search engine for open source code that extracts fine-grained structural information from the code. 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.

A Study of Ranking Schemes in Internet-Scale Code Search

Sushil Bajracharya*, Trung Ngo*, Erik Linstead†, Paul Rigor†, Yimeng Dou†,
Pierre Baldi†, Cristina Lopes*
*Institute for Software Research †Institute for Genomics and Bioinformatics
{sbajrach,trungcn,elinstea,prigor,ydou,pfbaldi,lopes}@ics.uci.edu

Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3423
ISR Technical Report # UCI-ISR-07-08
November 2007

Abstract. The large availability of source code on the Internet is enabling the emergence of specialized search engines that retrieve source code in response to a query. The ability to perform search at this scale amplifies some of the problems that also exist when search is performed at single-project level. Specifically, the number of hits can be several orders of magnitude higher, and the variety of conventions much broader.
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 results using some measure of relevance with respect to the terms being searched.
We present an assessment of 4 heuristics for ranking code search results. This assessment was performed using Sourcerer, a search engine for open source code that extracts fine-grained structural information from the code. 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 [26], 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 the 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, 3, 4, 5, 6]. 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 [30], code search is reported as the most common activity for software engineers. Sim et al. summarize a good list of motivations for code search [28] 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. Although Sourcerer can serve all the search goals mentioned, this paper focuses on the search for implementations.

But 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 *relevance*, so that the time that people spend searching for relevant code is minimized. The approaches taken by existing code search engines to the issue of ranking the results vary from conventional text-based ranking methods, to graph-based methods. From a research perspective, it would be desirable to conduct experiments that compare the performance of several existing search engines. That, however, has proven quite difficult, because they work on considerably different code bases (datasets). For example, the search for "quick sort" implementations in Java using Koders, Krugle, and Google codesearch, produces 636, 960, and 7,000+ results respectively. Moreover, the indexed code doesn't seem to be the same: for example, project JOElib, which contains an implementation of quick sort, seems to be

indexed by Koders and Krugle, but not by Google codesearch; conversely, the package www.di.fc.ul.pt/∼jpn/java/srcs/dataStructures.zip is indexed by Google codesearch, but not by the other two.

In order to focus on the issue of ranking code search results, we used Sourcerer to perform a comparative study of four schemes for ranking source code; the results are reported here. Access to the system, and all supporting materials for this paper, is available at `http://sourcerer.ics.uci.edu`.

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 7 we present related work. Finally Section 8 concludes the paper.

2 Sourcerer Architecture

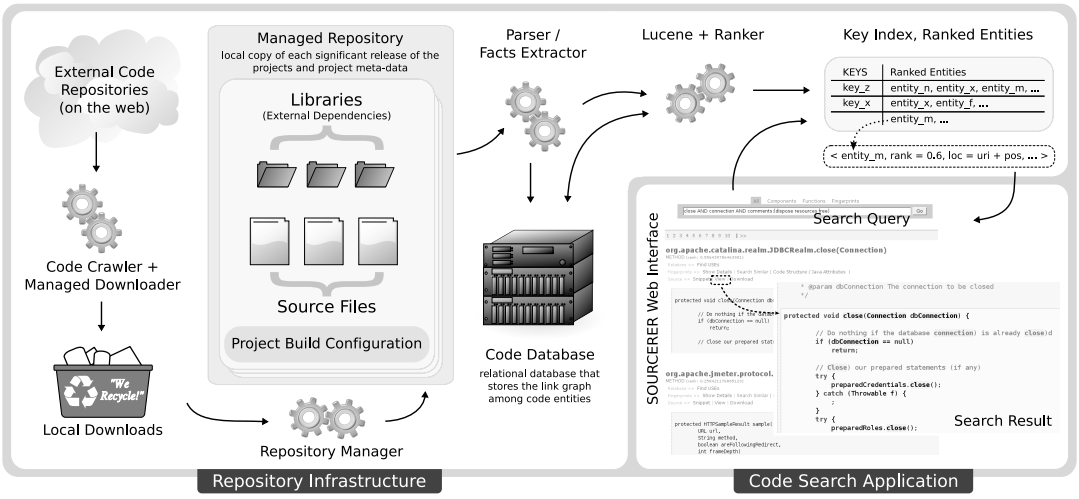


Fig. 1. Architecture of the Sourcerer infrastructure

Figure 1 shows the architecture of 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 Crawler and Managed Downloader:* We have implemented an extensible crawling framework that runs several kinds of crawlers as plugins: some target

well-known repositories, such as Sourceforge, others act as web spiders that look for arbitrary code available from web servers. The crawlers accumulate pertinent information about a project they find in the web. This information is used by a *Managed Downloader* to schedule the downloads (or checkouts) of source code from the projects' site to our local storage.

- *Repository Manager:* The local downloads of the projects are processed by the *Repository Manager* for some sanity checks such as presence of valid parsable content and libraries, and metadata extraction such as version information. Valid downloads are then added to the *Managed Repository*.
- *Managed Code Repository:* The system maintains a local copy of each significant release of the projects, as well as project specific meta-data. This meta-data is incrementally built starting with the information from crawlers to the information extracted from the parsers. For example, a project build configuration that conforms to to the project object model as given by the Maven build system `(http://maven.apache.org)`, is produced for each project in the repository using the dependency information obtained during parsing.
- *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. The parser was built using the JFlex `(http://www.jflex.de)` /CUPP `(http://www2.cs.tum .edu/projects/cup/)` toolset. Some highlights of the Parser 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.

With these components, we have an infrastructure for parsing and indexing large amounts of source code. This infrastructure is able to operate in various operating environments, ranging from personal desktop computers, used for development, to a cluster, used for indexing very large datasets. In the latter case, a custom task scheduler utilizes Suns Grid Engine 6 for parallel indexing of multiple repositories. We use a cluster of 80 computer nodes (Sun V20z) each with dual-core, dual-cpu AMD Opteron^{tm} Processor 250, and 4GB of RAM. Indexing over 4,000 projects amounting to 35 million SLOCs takes about 2.5 hours on this cluster.

Code Search Applications

We have developed two code search applications, one as a public web interface and the other as an internal tool specifically designed for performing the ranking experiments described in this paper. At their core, they both use the exact same mechanism for searching, which we explain in the next paragraph. Once the results are found, the web

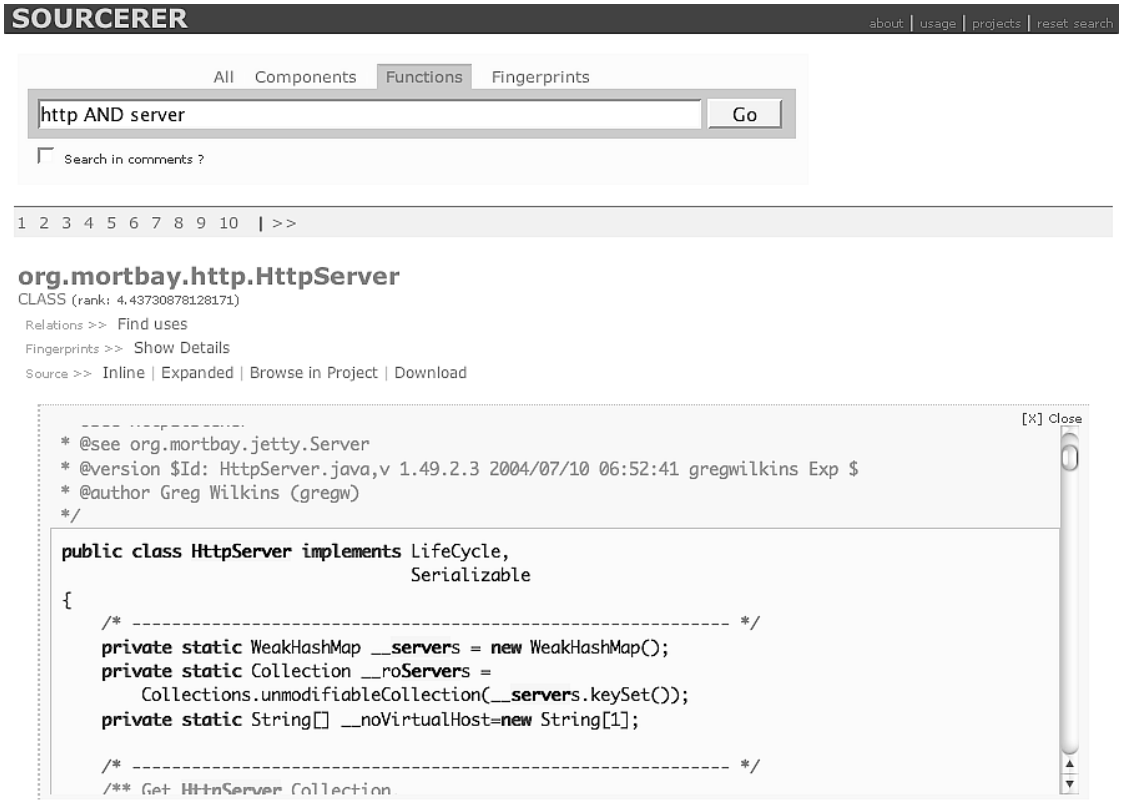


Fig. 2. User interface of the web search application, which can be accessed live at `http://sourcerer.ics.uci.edu`

application displays them along with information about the projects where they come from, and additional features (see Figure 2). The internal tool that we used for studying ranking takes an additional parameter consisting on the list of best hits for the control queries, and then outputs their positions in the retrieved results.

In either case, code search is built using the indexed keys and ranked entities in Sourcerer. It takes a list of query keywords entered by a user that are matched against the set of keywords maintained in the index (Lucene). Each key in the index is mapped to a list of entities. Each entity has its rank associated with it. Each entity also 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. 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 as of version 1.4 (we are not yet handling Java 5 features such as generics and annotations).

1. *Entities:* Entities are uniquely identifiable elements from the source code. Declarations produce unique program entity records in the database. Program entities can

[1] Relational models have been widely used for representing source code. See [21, 11, 10, 16], among others.

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) *holds*, a relation originating from a field to its declared type (Class or Interface), (5) *receives*, a relation originating from a method or a constructor to the type (Class or Interface) of any of its formal parameters, (6) *calls*, implying a source entity (method. constructor, or an initializer) makes a call on a target method or constructor, (7) *throws*, (8) *returns*, and, (9) *accesses*, a source entity (method, constructor or initializer) accesses a field.
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. A fingerprint is a sequence of numbers, implemented as a d-dimensional vector. This allows similarity to be measured using techniques such as cosine distance and inner product computation, well-known techniques in information retrieval [9].

   We have defined three types of fingerprints: (i) *Control structure fingerprint* that captures the control structure of an entity (ii) *Type fingerprint* that 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 (iii) *Micro patterns fingerprint* that captures information about implementation patterns, also known as micro patterns [15]. It contains 23 selected attributes from the catalog of 27 micro patterns presented in [15]. Fingerprints were not used in assessing the ranking heuristics, thus further discussion on them is out of the scope of this paper. Suffice it to say that they are used in Sourcerer to support structural searches of source code.

3.2 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 do not exist, then this cannot 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. In its simplest version, this pass consists of name matching between the referred library entities and source entities. However, there are some details that we need to account for, namely the fact that developers often reuse code by replicating the source of others project in their own projects – notice, for example, the results in Figure 2, where the same class org.mortbay.http.HttpServer appears in two different projects, jetty and usemodj. Moreover, Sourcerer has been designed to handle multiple versions of projects, as new releases are made available. As such, we have devised a probabilistic 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 *relevance* with respect to the terms being searched. We present the ranking methods that we have implemented and that are assessed in the next section.

Heuristic 1 (Baseline): TF-IDF on 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) [9]. TF-IDF's heuristic is that the relevance 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]

[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.

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 2: Packages, Classes, and Methods Only

When searching for implementations of functionality we can assume that those pieces of functionality are wrapped either in packages, classes, or methods, and aren't simply anonymous blocks of code. Moreover, most likely the developers have named those code entities with meaningful names. To capture the concept of "meaningful names" we used Fully Qualified Names for all entities in the code. 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 3: 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. As such, we weighted the results by boosting matches towards the right-hand side of names.

Heuristic 4: 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) [19], it has been suggested that a variation of Google's PageRank [20] 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 fine-grained than Spars-J's in that we apply it not only to classes but also to fields, methods and packages.
- 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): the dataset we used did not have significant amount of code replication.
- We exclude all classes from the JDK, because they rank the highest and developers know them well; as such, they are not that interesting for the purposes of the ranking assessment.

Additional comparisons between Spars-J and our work are given in Section 7.

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. We used a damping factor of 0.85, similarly to the original PageRank algorithm [20].

Id	Query Keywords* / Best Hits	# Hits
Q1	bounded + buffer	5
I	org.dbunit.util.concurrent.BoundedBuffer	
II	org.apache.turbine.util.pool.BoundedBuffer	
III	org.apache.commons.collections.buffer.BoundedFifoBuffer	
IV	org.apache.commons.collections.BoundedFifoBuffer	
V	org.apache.commons.collections.buffer.BoundedBuffer	
Q2	quick + sort	10
I	cern.colt.Sorting.quickSort(Object[],int,int,Comparator)	
II	de.teamwork.util.QuickSort	
III	org.apache.forrest.forrestdoc.java.src.util.QuickSort .quickSort(Object,int,int,Comparator)	
IV	org.tzi.ideal.textutils.FastQuickSort	
V	org.htmlparser.util.sort.Sort.QuickSort(Ordered[])	
VI	ca.qc.cslaval.agenda.util.QuickSort	
VII	org.apache.turbine.util.QuickSort.quickSort(Object,int,int,Comparable)	
VIII	org.htmlparser.util.sort.Sort.QuickSort(Vector,int,int)	
IX	org.htmlparser.util.sort.Sort.QuickSort(Sortable,int,int)	
X	de.unicats.agents.integrationAgent.kWaySorting.QuickSort	
Q3	depth + first + search	5
I	Mtdf.spawn_depthFirstSearch(NodeType,int,int,short)	
II	jopt.csp.spi.search.technique.DepthFirstSearch	
III	org.mojave.jeat.search.dfs.BasicDepthFirstSearch	
IV	org.mojave.jeat.search.dfs.BranchAndBoundDepthFirstSearch	
V	org.openscience.cdk.graph.PathTools.depthFirstTargetSearch (AtomContainer,Atom,Atom,AtomContainer)	
Q4	regular + expression	3
I	org.apache.xerces.impl.xpath.regex.RegularExpression	
II	org.apache.xmlbeans.impl.regex.RegularExpression	
III	org.apache.tools.ant.types.RegularExpression	
Q5	tic + tac + toe	3
I	org.lnicholls.javahmo.plugins.games.TicTacToe	
II	com.levelonelabs.aimbot.modules.TicTacToeModule	
III	TicTacToe	
	Total Best Hits :	57

Id	Query Keywords* / Best Hits	# Hits
Q6	ftp + server	5
I	org.tzi.ideal.ftp.IdealFtpServer	
II	org.mortbay.ftp.TestServer	
III	photoorganizer.ftp.Ftp.port(ServerSocket)	
IV	org.apache.jmeter.protocol.ftp.sampler.FTPSampler.getServer()	
V	org.carion.s3dav.ftp.FTPServer.stopServer()	
Q7	tcp + server	10
I	EppTestServerTcp	
II	ibis.impl.nameServer.tcp.NameServer	
III	com.indigotp.is.TcpServer	
IV	de.siemens.fast.core.container.plugin.transfer.net.AbstractServerTCP	
V	ibis.impl.nameServer.tcp.NameServerClient	
VI	thera.servers.TcpServer	
VII	thera.servers.MultithreadTcpServer	
VIII	org.apache.jmeter.protocol.tcp.sampler.TCPSampler.setServer(String)	
IX	EppTestServerTcp.main(String[])	
X	org.apache.tomcat.util.net.PoolTcpEndpoint .setServerSocket(ServerSocket)	
Q8	rmi + server	5
I	net.sf.crispy.impl.rmi.MiniRmiServer	
II	uk.midearth.dvb.server.RMI	
III	marquee.xmlrpc.objectcomm.example.XmlRmiServer	
IV	nz.co.vil.mp3.rmi.DBRemoteServer	
V	consciouscode.seedling.server.RmiRegistryNode	
Q9	chat + server	7
I	com.bribble.nl.Core.FlashChatServer	
II	KEWLChat.server.ChatServer	
III	myclass.chat.ChatServer	
IV	net.sourceforge.jsrvany.test.ChatServer	
V	net.sourceforge.jsrvany.test.ChatServerThread	
VI	echo2example.chatserver.ChatServerServlet	
VII	ChatServer	
Q10	ftp + client	4
I	com.indigotp.ftp.FtpClient	
II	com.visualmetrics.biostream.ftp.FtpClient	
III	org.apache.jmeter.protocol.ftp.sampler.FtpClient	
IV	org.osdk.ftp.enterprisedt.EdtFtpClient	

Fig. 3. Control queries used in the assessment of Sourcerer showing the keywords used and the best hits hand-picked by human oracles for each query. ('+' in Keywords implies logical AND)

5 Methodology for Assessing Ranking Heuristics

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

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.

The dataset used for this study consists of 1,555 open source Java projects (see Fig. 6, at the end of the paper), corresponding to 254 thousand classes and 17 million SLOCs.

Comparisons between Sourcerer and other search engines are problematic, because there is no benchmark dataset, as discussed before. We note that other work on source code search engines has included comparisons with Google general search (see e.g. [19]). That is not controlled enough to address the assessment goals we are pursuing. We believe that the methodology we devised, described in this section, is the most appropriate for the goal at hand. Our assessment methodology is comparable to that used in [23]. We are making the dataset used in this study publicly available, so that other approaches can be compared to ours.

5.1 Assessment Goal

The goal of the assessment is to compare the text-based, structure-based, and graph-based methods described in Section 4, and combinations of those methods. 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 a first step into the answer.

5.2 Control Queries

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. A group of five expert Java developers were involved in collectively selecting the best hits for each query. The following criteria were used to select the "Best 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

We defined 10 control queries (see Fig. 3) to be run against an indexed code repository described above. 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).

Figure 3 shows all 10 control queries, the combination of keywords used for each control query and the best hits as ranked by the human oracles. The ranks are indicated by roman numerals. In total for the 10 control queries issued 57 search results were selected as the best hits.

5.3 Ranking Schemes

Five ranking schemes were derived from the four heuristics described in Sec. 4, some corresponding to combinations of the heuristics:

1. *Baseline*: this corresponds directly to Heuristic 1.
2. *FQNs*: this corresponds directly to Heuristic 2.
3. *FQNs + coderank*: this corresponds to a combination of Heuristics 2 and 4. Implementing this ranking scheme involves combining Lucene-based TF-IDF scores with the computed coderank (Heuristic 4) values to produce a total ordering of results. To achieve this, search results are first placed in order of decreasing TF-IDF score. Those ranked results are then partitioned into bins consisting of k results each, and each bin is in turn sorted based on coderank. The net effect is to give precedence to highly relevant hits based on content, with coderank acting as a secondary, popularity-based ranking refinement to reorder documents with similar TF-IDF scores.
4. *FQNs + right-hs boost*: this is a combination between Heuristics 2 and 3, which employs Specificity (Heuristic 3) to improve the uniform TF-IDF approach of Heuristic 2. The standard TF-IDF scheme is augmented to include explicit weights on fields in the document index. Commonly referred to as field "boosting" in the text retrieval domain, this approach allows search results with matching query terms in specified fields to be given a higher score than results with matching query terms in other fields. In this ranking scheme, results with query term matches in the rightmost part of the fully qualified name are given a higher score than results with matching query terms in FQN positions further to the left. Because boosting is integrated into the TF-IDF scoring process (Lucene, in our case), it is not necessary to bin the results for a secondary ranking pass.
5. *All*: the final ranking scheme corresponds to a combination of all (non-baseline) ranking heuristics, i.e. it combines boosted TF-IDF on FQNs with coderank. In this scheme, results are first ordered based on the boosted TF-IDF scores described above; coderank is then applied in a second ranking pass using the binning technique described for scheme 3.

5.4 Comparison of Ranking Schemes

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.

6 Results

Ranking scheme	*Top 10*	*Top 20*
1. Baseline (code-as-text)	30%	44%
2. FQNs	32%	42%
3. FQNs + coderank	40%	44%
4. FQNs + right-hs boost	63%	74%
5. FQNs + coderank + right-hs boost	67%	74%

Table 1. Recall of best hits within top positions – top 10 and top 20 positions shown.

Table 1 summarizes the results of our study. Figure 4 shows a visual representation of these results. The plot in Figure 4 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 major observations from Table 1 and Figure 4 are the following: (1) there is no noticeable difference between schemes 1 (baseline, full-text indexing) and 2 (FQNs only); and (2) the most noticeable improvement with a single heuristic lies in the introduction of specificity (scheme 4). We expand on these observations with more detailed data.

Figure 5 shows all the data from the study. The first detail to notice is the differences in query formulation between the baseline and the other schemes. For example, the query for searching a bounded buffer implementation was formulated as "boundedbuffer* OR (bounded* AND buffer*)" for the baseline, and simply "bounded AND buffer" in the other schemes. In the case of code-as-text, we have to formulate the queries at that lower-level, almost as direct string match (case is ignored in Lucene), because no naming conventions are being considered. In general, for the baseline measurements, code-as-text with Lucene, query (*term*) is formulated as (*term**), query ($term_1$ $term_2$) is formulated as ($term_1 term_2$ *OR* ($term_1$ * *AND* $term_2$ *)), etc. For all the other schemes, and given that term unfolding is done during parsing and indexing according to some basic Java naming guidelines, query (*term*) is simply (*term*), query ($term_1$ $term_2$) is ($term_1$ *AND* $term_2$), etc.

The two formulations don't fully match, in the sense that they don't retrieve the exact same records. That is visible in columns marked as "A" in Figure 5, where the number of hits for the baseline is different than for all the other schemes. For example, for Q2 (quick sort), the baseline retrieves more than twice the results retrieved by

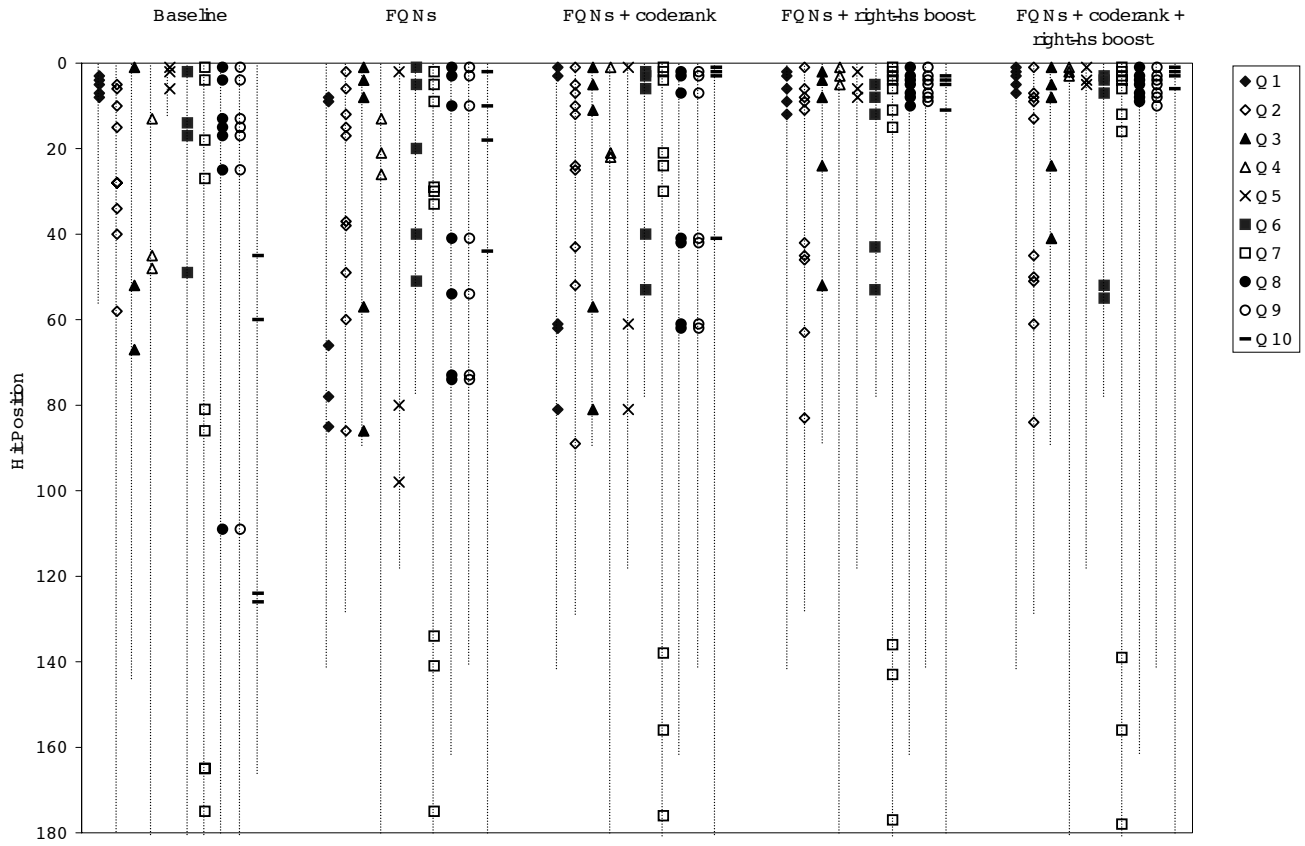


Fig. 4. Placement of best hits by the different ranking schemes.

			A		B										C	
					Ranks from the Best Hits											
	Id	Queries	# of Hits	Recall	I	II	III	IV	V	VI	VII	VIII	IX	X	**N = 10**	*N = 20*
Baseline	Q1	boundedbuffer* OR (bounded* AND buffer*)	56	1.00	**3**	**4**	**5**	**7**	**8**						5	5
	Q2	quicksort* OR (quick* AND sort*)	262	1.00	**5**	**6**	**10**	*15*	28	28	28	34	40	58	3	4
	Q3	depthfirstsearch* OR (depth* AND first* AND search*)	144	0.60	**1**	52	67								1	1
	Q4	regularexpression* OR (regular* AND expression*)	748	1.00	*13*	45	48								0	1
	Q5	tictactoe* OR (tic* AND tac* AND toe*)	12	1.00	**1**	**2**	**6**								3	3
	Q6	ftpserver* OR (ftp* AND server*)	186	0.80	**2**	*14*	*17*	49							1	3
	Q7	tcpserver* OR (tcp* AND server*)	355	0.90	**1**	**4**	*18*	27	81	86	165	165	175		2	3
	Q8	rmiserver* OR (rmi* AND server*)	237	0.80	25	61	92	96							0	0
	Q9	chatserver* OR (chat* AND server*)	212	1.00	**1**	**4**	*13*	*15*	*17*	25	109				2	5
	Q10	ftpclient* OR (ftp* AND client*)	167	1.00	45	60	124	126							0	0
		Total entities from the Best Hits captured in all queries within N positions in the results													**17**	*25*
		Recall for Top N Hits													**0.30**	*0.44*
FQNs	Q1	bounded AND buffer	141	1.00	**8**	**9**	66	78	85						2	2
	Q2	quick AND sort	129	1.00	**2**	**6**	*12*	*15*	*17*	37	38	49	60	86	2	5
	Q3	depth AND first AND search	86	1.00	**1**	**4**	**8**	57	86						3	3
	Q4	regular AND expression	193	1.00	*13*	21	26								0	1
	Q5	tic AND tac AND toe	118	1.00	**2**	80	98								1	1
	Q6	ftp AND server	77	1.00	**1**	**5**	*20*	40	51						2	3
	Q7	tcp AND server	206	1.00	**2**	**5**	**9**	29	30	33	134	141	175	187	3	2
	Q8	rmi AND server	162	1.00	*11*	42	63	107	122						0	1
	Q9	chat AND server	141	1.00	**1**	**3**	**10**	41	54	73	74				3	3
	Q10	ftp AND client	201	1.00	**2**	**10**	*18*	44							2	3
		Total entities from the Best Hits captured in all queries within N positions in the results													**18**	*24*
		Recall for Top N Hits													**0.32**	*0.42*
FQNs + coderank	Q1	bounded AND buffer	141	1.00	**1**	**3**	61	62	81						2	2
	Q2	quick AND sort	129	1.00	**1**	**5**	**7**	**10**	*12*	24	25	43	52	89	4	5
	Q3	depth AND first AND search	86	1.00	**1**	**5**	*11*	57	81						2	3
	Q4	regular AND expression	193	1.00	**1**	21	22								1	1
	Q5	tic AND tac AND toe	118	1.00	**1**	61	81								1	1
	Q6	ftp AND server	77	1.00	**2**	**3**	**6**	40	53						3	3
	Q7	tcp AND server	206	1.00	**1**	**2**	**4**	21	24	30	138	156	176	181	3	3
	Q8	rmi AND server	162	1.00	**1**	44	63	102	129						1	1
	Q9	chat AND server	141	1.00	**2**	**3**	**7**	41	42	61	62				3	3
	Q10	ftp AND client	201	1.00	**1**	**2**	**3**	41							3	3
		Total entities from the Best Hits captured in all queries within N positions in the results													**23**	*25*
		Recall for Top N Hits													**0.40**	*0.44*
FQNs + right-hs boost	Q1	bounded AND buffer	141	1.00	**2**	**3**	**6**	**9**	*12*						4	5
	Q2	quick AND sort	129	1.00	**1**	**6**	**8**	**9**	*11*	42	45	46	63	83	4	5
	Q3	depth AND first AND search	86	1.00	**2**	**4**	**8**	24	52						3	3
	Q4	regular AND expression	193	1.00	**1**	**3**	**5**								3	3
	Q5	tic AND tac AND toe	118	1.00	**2**	**6**	**8**								3	3
	Q6	ftp AND server	77	1.00	**5**	**8**	*12*	43	53						2	3
	Q7	tcp AND server	206	1.00	**1**	**2**	**3**	**4**	**6**	*11*	*15*	136	143	177	5	7
	Q8	rmi AND server	162	1.00	**1**	**2**	46	66	109						2	2
	Q9	chat AND server	141	1.00	**1**	**3**	**4**	**5**	**7**	**8**	**9**				7	7
	Q10	ftp AND client	201	1.00	**3**	**4**	**5**	*11*							3	4
		Total entities from the Best Hits captured in all queries within N positions in the results													**36**	*42*
		Recall for Top N Hits													**0.63**	*0.74*
All	Q1	bounded AND buffer	141	1.00	**1**	**2**	**3**	**5**	**7**						5	5
	Q2	quick AND sort	129	1.00	**1**	**7**	**8**	**9**	*13*	45	50	51	61	84	4	5
	Q3	depth AND first AND search	86	1.00	**1**	**5**	**8**	24	41						3	3
	Q4	regular AND expression	193	1.00	**1**	**2**	**3**								3	3
	Q5	tic AND tac AND toe	118	1.00	**1**	**4**	**5**								3	3
	Q6	ftp AND server	77	1.00	**3**	**4**	**7**	52	55						3	3
	Q7	tcp AND server	206	1.00	**1**	**2**	**3**	**4**	**6**	*12*	*16*	139	156	178	5	7
	Q8	rmi AND server	162	1.00	**1**	*15*	42	63	102						1	2
	Q9	chat AND server	141	1.00	**1**	**3**	**4**	**5**	**7**	**8**	**10**				7	7
	Q10	ftp AND client	201	1.00	**1**	**2**	**3**	**6**							4	4
		Total entities from the Best Hits captured in all queries within N positions in the results													**38**	*42*
		Recall for Top N Hits													**0.67**	*0.74*

Fig. 5. Calculation of Recall values for each ranking scheme within top N hits in the results

the other schemes. A close observation of the extra matches of the baseline showed that the vast majority of the extra hits were noise – the words "quick sort" being mentioned in comments (e.g. a comment "We are using quick sort for sorting this list" over the implementation of something else). Conversely, some queries in the baseline miss some the identified best hits. For example, the formulation of Q3 (depth first search) in the baseline missed two of the best hits. This was because of a limitation of Lucene, which doesn't allow wildcards in the beginning of terms – therefore missing hits such as "spawn_depthFirstSearch."

In any case, the two different query formulations are close enough for the baseline to have the similar total *recall* as the others: in the baseline measurements, only 5 out of 57 best hits were not retrieved. As such, it is valid to compare the baseline experiment with the others, since (a) we are looking at where the schemes place the best hits, and 91% of the identified best hits are retrieved by the baseline; and (b) the extra results provided by the baseline are essentially noise, i.e. no new relevant results were identified there.

Columns marked as "B" in Figure 5 show the placement of the best hits in the search results for each ranking scheme. For example, in the baseline, Q1 (bounded buffer), the 5 best hits were placed in positions 3, 4, 5, 7, and 8 of the list of 56 results.

Finally, columns marked "C" are used to compute the recall of best hits within the first N positions. For example, in the baseline, Q2 (quick sort) there were 3 best hits within the first 10 positions – namely in positions 5, 6, and 10 – and only one additional best hit in the top 20 positions – the hit in position 15. All other were above position 20. The recall values within the top 10 and 20 positions for each ranking scheme were calculated by adding the corresponding columns above and dividing that number by the total number of best hits identified by the human oracles (57 – see Figure 3).

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).* One of the most interesting results of this study is that the performance of schemes 1 and 2 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 fairly irrelevant for the search purposes. As noted above, using code-as-text often results in extra hits, usually found in comments or in method bodies, that proved to be irrelevant for the searches at hand. This result indicates that semantic search engines can safely reduce the amount of indexed information by ignoring a large amount of textual information.

  A production engine where this result might be applicable is Google code search [6], which treats code as text. For example, when searching for an http server implementation in Java using Google code search, we note that most of the good hits (i.e. actual http server implementations) correspond to classes whose names include "http" and "server." The extra possible matches, which can be seen with queries such as "http AND server lang:java," are essentially noise, such as occurrences of these words in comments (e.g. http//:...).

  This observation has a qualitative explanation in the way that developers are taught to name code entities. Careful naming has traditionally been considered a good

software practice, and it looks like developers are applying those guidelines [7] in practice.

This also reflects our design decision of using FQNs as the name space to search. We have observed that developers often rely on containment hierarchies to assign semantics to the names they decide to give to a certain program entity. This has some important consequences:

1. The exact name of the entity as it appears in its definition in the source code text may not contain all the query terms a user will use to search for that entity. For example, in Fig. 3 the second best hit for query Q6 (ftp + server) is an entity named "org.mortbay.ftp.TestServer". The simple name of this entity is "TestServer". Relying only on the simple name of this entity will not associate the term "ftp" with it and thus it will not be found when a user searches for "ftpserver." A lookup in Google code search ("testserver ftp mortbay lang:java") indicates that their engine is indexing this entity, but that it is not retrieved on the query "ftp server." In the total 57 best hits, there are 11 hits that share this property. The 2nd best hit for Q7 and 2nd best hit for Q8 are two noticeable ones.
2. Using FQNs as the meaningful names can boost the significance of an entity. Since we are using TF-IDF measures for ranking, the frequency of occurrence of the term effects the ranking. It can be seen that using FQN increases the frequency of relevant terms in the name. For example, in Fig 3, the 3rd best hit for query Q2 has the FQN "org.apache.commons. collections.buffer.BoundedFifoBuffer." Using the FQN makes the term "buffer" appear twice in the name. Twenty out of 57 best hits share this property.
3. The prefix that gets appended to the exact name associates some meaningful terms with the entity that reflects the hierarchic decomposition that the developers have encoded in the implementation. For example, in Fig. 3, the 1st best hit for Q5 has a term "games" associated with it. A user who is searching for "tic + tac + toe" probably might refine the query with an additional term "game."

These observations help us further to understand the effect of coderank and specificity in ranking search results.

- *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 only 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. This result somewhat challenges a previous result reported by Inaoue et al. [8, 18, 19], in which they found strong benefits in using a similar popularity-based technique. We will expand on this in the next section.
- *Effect of specificity (FQNs + right-hs boost).* This heuristic (specificity) is the one that has the strongest positive effect on the results. In order to understand this, we need to refer to our design decision of using FQNs. The use of FQNs as the namespace to search adds some relevant terms as discussed earlier, but, it also has the side effect of associating less relevant terms, some merely noise, with the entities. Less

specific terms from the FQNs that usually are found towards the left (such as "org," "com," "cern," "apache" etc) do not tell much about the entity they are associated with. In most of the cases, the hierarchic nature of the FQN namespace makes terms in right part of the FQN more specific. By boosting the terms found at the rightmost of the FQNs, the specificity heuristic avoids pulling irrelevant entities, that might have the query term appearing somewhere to the left side of the FQN, to the top of the results.

In short, taking into account the hierarchical decomposition of the program's namespace enables the search method to place more relevant entities in the top positions of the result set.

– *Combinations.* Overall, the ranking scheme that performed better was the one that includes a combination of text-based, structure-based, and graph-based heuristics. The individual contributions of the different heuristics, even if small, have an additive effect. We believe that further improvements will be obtained by devising additional heuristics that can then be combined, in a weighted manner, with the ones described here.

7 Related Work

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

7.1 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 [20] 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 [3], Codase [4], csourcesearch [5], and Google Code Search [6] are the few prominent ones. All of these code search engines seem to be crawling publicly available code in the Internet and are almost functionally equivalent with features such as: allowing search for multiple languages, allowing search in multiple fields in code (such as comments, definitions of entities), basic filtering mechanisms for results (based on types of entities, licenses) etc. Google code search has an

additional feature of allowing a regular expression search on the code. Regular expressions can be quite powerful in expressing queries but, beyond this, Google code search does not seem to implement any notion of structure-based search and ranking of results incorporated so far. It still is a string-based regular expression match over a massive collection of plain source code text.

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 [8, 18, 19] 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 relevance 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 obtained here, 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 relevance. 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 [25] 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.

7.2 Software Engineering Tools

Modern software engineering tools are bringing more sophisticated search capabilities into development environments extending their traditionally limited browsing and searching capabilities [17, 22, 29, 24]. 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 [22]. 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 [17]. 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 [29] and JIRiss [24] 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 [23]. Both these tools lack graph based techniques and thus are limited to their specific IR specific ranking in presenting results.

XSnippet is a code assistant system that assists developers in finding relevant code snippets [27]. Like Stratchoma it uses the developer's current context to generate queries and uses a graph-based model of the source code to mine relevant snippets in a code repository. The snippets can be ranked according to four different types of ranking heuristics. While *XSnippet* strengthens the claim that structure-based ranking is a significant contributor to the usability of the returned results in searching for code snippets in a repository, it suffers from two limitations. First, the types of queries are limited to looking for solutions for object instantiations only. Second, the very nature of the snippet mining algorithm requires the system to traverse the code graph in the repository for each query. This restricts the use of the technique on any large scale repository like Sourcerer.

7.3 Structure-based Source Code Search

Recent work as presented in [12, 31, 16, 14] captures the state-of-the art in using the relational model and efficient representation (such as datalog) for practical extraction of structurally related facts from source code bases. We are exploring ways to scale these techniques to the level of Internet-scale code search using Sourcerer.

8 Conclusions

We have presented Sourcerer, a search engine for open source code search built that extracts fine-grained structural information about source code. In the process of developing Sourcerer, we have been experimenting with several ranking heuristics, as well as with assessment methodologies that can tell us whether those heuristics are useful or not. In this paper, we described four such heuristics and showed their performance using our assessment methodology. The first important result is the realization that most of the semantic information is in the names of code entities. As such, source code search engines can safely reduce the amount of information that needs to be indexed. Beyond

that, the heuristic that showed the highest improvement in relevance 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 ranking the results of 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 are 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 the search capabilities of 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] Krugle web site. *http://www.krugle.com*.

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

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

[6] Google Code Search. *http://www.google.com/codesearch*.

[7] Sun Microsystems. Code Conventions for the Java Programming Language, *http://java.sun.com/docs/codeconv/html/ CodeConvTOC.doc.html*.

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

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

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

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

[12] Tal Cohen, Joseph (Yossi) Gil, and Itay Maman. Jtl: the java tools language. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 89–108, New York, NY, USA, 2006. ACM Press.

Projects from sourceforge.net													*apache.org*
abbot	burrokeet	ddraig	fe2	hydra-mcmc	jbrowser	jphotar	logtail	netant	petra	romaframework	streamingmedia	vocabulala	beehive_release
abes	c2h	debate-ie-timer	ffs3dui	hydrate	jbs	jplus	lomagcvs	netftracker	pfplms	rommanager	streammp3	vraptor	beehive_src
abora	c2nak	decodebase64	fibs	iafw	jbsse	jpoller	lsketch	netmail	pfp-studio	rox-xmlrpc	stringulator	vt6530	cocoon
absim	c3p0	deeparc	filebunker	ibrowse	jcas	jportforwarder	ltoolbar	netmessenger	pharmacy	rpg-chat-server	struktor	vtd-xml	db_ddlutils
accs	c4j	defectspy	filehashler	ibsxnet	jeckit	jpos	ltsa	netpop	phoenixjms	rrdtoolbox	strutsback	vtsurvey	db_derby
acctsync	cajax	deforex	filescatcher	icap-server	jceps	jpowergraph	lucidium	netstereo	phono	rssviewer	strutstest	walter	db_jdo
acdesigner	calamar	delambre	filesink	ichabod	jcfd	jproactor	lucrative	networktools	photoarch	rt-com	subnet	wapide	db_ojb
aceoperator	camelseye	delicon	finaldestiny	ichatle	jcharmanager	jpsql	lunchtime	networth-java	photo-lith	rtpdemo	subsonic	wapmon	db_torque
aclocator	canvasstudio	delphiblue	findclass	icpr0n	jchatserver	jrail	lutsko	nexsm	photoorg	rtp-text-t140	sunxacml	wapreview	directory_mina
acscriptworks	carbonfive	demetrix	firstclient	identity	jcid	jrar	luxor-contrib	nextmock	phpjrkdb	rubyvm	superficie	warping	excalibur
active-jms	carpincho	detach	fitdecorator	iform	jclasslib	jrate	lxb4j	nfop	phplabdb	rucpyc20	support-kbase	waw	forrest
adi	casanova	devon	flashjava	ifx-tx-switch	jclate	jrdf	m2c	nicq	php-mailer	rvchat	surfraw	webacct	bcel
aegisvm	caswp	devtimetracker	flatworm	ikvm	jcolorgrid	jreplugintest	m6800	nino	phpmp3	rvngofthe5kngdm	svnformat	webcamapplet	bsf
afp-renderer	catapultframewk	dewdrop	fling	ili2sql	jconpool	jrevpro	madhavi	njs	phpmybibli	s2s	swg-crafter	web-cat	cactus
agel	catmdedit	dghmux-java	fll-sw	imageapp	j-controller	jrobot	mafra-toolkit	northstarbbs	phpmychord	s3dav	swgman	webcockpit	commons_attributes
agentcell	cazare	dgpf	flowrider	imagemover	jcq2k	jsane-net	magictool	nosleepsoftware	phpmyscs	sablecc	swingbean	webdialer	commons_beanutils
agragator	cb2xml	dguitar	foedus	img-browse-sort	jcrontab	jsch	magma	nphp	phpvideopro	safejdbc	swingosc	webdict	commons_betwixt
ajax	ccsh	dhtmlwindowmngr	foogoo	im-java	jcryptfx	jsci	mailfetcher	nplayer	phpworldportal	saffron	swlink	webfem	commons_chain
ajc-for-eclipse	cdctl	dhvani	formhammer	immajellan	jdbc2jdbc	jscl-meditor	mailfw	nrl	physhun	sag	swlpf	webjabber	commons_cli
alan	cdk	dialup	forumromanus	impact	jdbcexplorer	jscorekeeper	mailremover	nukechat	pi4soa	salwalib	sybtool	weblayout	commons_codec
alfaqtiengine	cdlib	digijsim	fplus	importwizard	jdbcimporter	jscp	mars-sim	numbat	picture-applet	samjr	synclast	webmounty	commons_collections
amazed	cdphotoindex	digitalworkroom	fracas	imshell	jdblogger	jsdesigner	math4j	nups	picturebook	samurai-kanji	t4bi-kf	webpresenter	commons_configuration
ammentos	cehnet	dipper	fractalfabric	infinitypfm	jdbforms	jsdsi	matheclipse	nwntools	pinghealth	sat3c	takocompiler	webreplay	commons_daemon
ana-mp	celestjava	directorysync	fraid	infojection	jdbm	jsjabber	mathpixel	oaiarc	pinkracoon	sbdaemon	ta-lib	webtheme	commons_dbcp
anastacia	centropub	diskdom	framerd	intermezzo	jde-usages	jsman	maverick	oaikepler	pipe2	schemacrawler	tankcombat	wenbozhu	commons_dbutils
anathema	cfmximgateways	distributeddns	frameworkatg	intpasmga	jdforum	jsmbconf	mbfuzzit	oasisgame	pisces	sciml	tara	wequest	commons_digester
anaxagora	chain	dita-ot	freearc	invoicex	jding	jsnpp	mbmdotnet	obfunae	pisdnmon	scoja	tasktrack9	whale-sqlfilter	commons_discovery
angst	chainedoptions	div-dev	freebbs	ipdump	jdocfilter	jsoapserver	mbootp	obix	pittrainer	scribesw	tassel	wibs	commons_el
animix	chartfactory	diy-zoning	freeforums	ipgrab	jdots	jspcontrols	mdevinf	obix-framework	pixe	scts	taste	wicket-stuff	commons_email
anis	chartpart	djdoc	freemetal	ipony	jdr3d	jspim	mdm	objectbuilder	pjltella	sdd1	taxgeek	wife	commons_fileupload
annotatio	chatinjava	dlbook	freeserver	ipr	jdraw	jspmsn	meddix	objectcanvas	plarpebu	sdtraffic	taxmadad	wigs	commons_httpclient
ant	chemnomparse	dmabco	freetts	ircappender	jdrawing	jsqlbuilder	mediachest	observation	plateau	secure-webmail	taxogen	wikicpp	commons_id
antifirewall	children-s-shop	dmdf	freewebchat	ispc	jdvb	jsqlparser	mediaframe	oci8py	pldoc	seedling	tbug	wiskee	commons_io
antsp2p	chipchat	dmz	frexplore	ispheres	jedpoint	jstimpy	medlane	octet	plexone	sehr	tclreadline	witl	commons_jelly
aoemudataloggin	chordcast	doblet	frood	ispman	jeeg	jstyx	memoranda	office-manager	plop	selftest	tcpbeholder	wnjn	commons_jexl
aojava	chronus	doco	funambolwapmail	isys2	jeepproject	jsvalidator	memorize-words	off-sync	plotlib	sendmail-jilter	team-assistant	workorders	commons_jxpath
applecommander	cinc	dogwood	funcplotter	itunesanywhere	jena	jsvc	memoryjndi	oggcarton	pluk	sendtrap	teaseme	woweapons	commons_lang
apvision	cinter	doksproject	functionalj	ivjmud	jepla	jsvector	meow	ohla	plutostatus	seplus	ted	wphotomarket	commons_launcher
araneaframework	cjos	dol	funtoosh	j2dcg	jert	jsvg	merd	ojbc	plwinmacro	servletspy	teha	wsvjdbc	commons_logging
arara	cli4j	domingo	fvframework	j2eeadapters	jete	jsvm2	messageit	ojob	pmii	sfdd	telnetoverhttp	wtt	commons_math
arbat	click	donut	fxl-project	j2mefs	jevent	jswiki	metacoretex	olitext	pmlibs	sfljtse	terminator	wwbota	commons_modeler
arbre	clipboardtodb	draw	galapago	j2mesafe	jexecutor	jsxmldoc	metadatamanager	ome	pnx	sfutils	tesijava	wwportlet	commons_net
archive-crawler	clixml	drawswf	galena	j2mewordml	jexp	jsynoptic	metafileman	omegachat	pocketbook	shame	texidoclet	wxjava	commons_pipeline
arise	cmusphinx	dreamboxx	galleon	j2s	jextractor	jtbrpg	metas	ominit	polepos	shanidom	textbender	x10controller	commons_pool
art-testbed	cobra	drinkmixer	gameborg	j4rcs	jfacedbc	jtestcase	metavnc	omnigene	pondscum	shift	tfn	xagentx	commons_primitives
asa-atl	cocchargen	dropandsend	ganimus	jabberbeans	jfigure	jtexsynth	mhptester	omnihelp	portablog	shift2ingres	thea	xbel4ie	commons_proxy
asdn	coffeemud	dropboxmq	ganymedssh2	jacii	jfin	jthumbs	miamexpress	onlinecompiler	portal-engine	sia	theartbeat	xbone	commons_validator
asm2class	cogunity	ds3	ganzua	jackcess	jflightlog	jtidy	micro-jabber	oo-widgets	portecle	siegeweb	thefix2	xbrlapi	commons_vfs
aspectj4netbean	colander	dsaman	garchiver	jacomma	jfred	jtimetracker	micsimkit	opcion	portletunit	signstream	theopenerp	xbrlcore	ecs
asse	colt	dspro	gata	jador	jfreechart	jtkinter	midi-coach	open2dprot	ppts	sikher	thop	xbrowser	hivemind
asterfax	comicreader	dubman	gatman	j-a-g-a	jfreedesigner	jtrains	midiquickfix	openaccess	praya	silence	thout	xbus	httpcomponents
astroalgorithms	comicsfetcher	ducktail	gattmath	jagents	jgabl	jtreemap	mightycal	openantivirus	presentingxml	simasp	thundernote	xc4j	jmeter
astroinfo	comicviewer	duine	gazelle	jagorobots	jgames	juddi	mijal	openbns	privateaccessor	simgear	tigerunit	xdc	oro
atalk	comitatuscommon	dukebot	gccirc	jaiviewer	jgantt	juipiter	millebv	opencap	probonophp	simloc	tijdr	xdprof	poi
atcj	compframework	dungasync	gcms	jajuk	jglade	jukex	millscript	openefm	processdash	simple-inside	tikeswing	xdvr	regexp
auctionmage	complat	dynamic-vc	gdyntext	jalgorithm	jgnuplot	jundo	mipitanza	openfem	proclog	simpleinvoices	timeentry	xenonbbs	slide_server
autoopencas	comtorj2me	e8x2	gebora	jaligner	jgrabber	jung	mlang	openflow-mag	progdown	simple-request	timetowork	xgen	slide_webdavclient
avantgarde	contex	earutils	gedcomfilter	jalingo	jgrapht	junimedia	mlcrosswords	openforecast	prologpluscg	simpletracker	tiniedcar	xgl-to-jogl	taglibs
averniasoc	controle	easter	genex	jamato	jguard	jurpe	mmitethys	openhardware	protege-docgen	simulatron	tipcon1	xgql	tapestry
avidb	conv	easyaccept	genj	jam-daq	jgv	jusecase	mmix-mode	openi	protein3dprint	simuledufg	titulus	xicesdk	tomcat
awesum2	cook	easycompile	geoapi	jamod	jhlib	justjournal	mmmysql	openipmp	proteinmusic	sinaxe	tladmin	xing	turbine
axis-wsse	coppercore	easy-dev	georaptor	jamper	jhomenet	jux	mmopengraph	openjacc	prowser	sinc	tmlight	xletview	velocity
azautopilot	coras	easypersistence	geotiff-jai	janissary	jhoney	jvclifford	mms	openjgroup	psh	sino	tmpl	xlsjdbc	lenya
azweb	corbaerp	eatool	gfd	janitor	jhotdraw	jveda	mobilebuddy	openjms	psretprojects	sjpt	toastscript	xml2db-eai	lucene_hadoop
b2bframework	corbatrace	ebaybiddertool	ghostlink	jano	jib	jvending	mobix	openlabel	pulga	sjs	tonermeter	xmlgraph	lucene_lucene
b4browser	cornucopia	ebjava	gjal	janu	jimm	jvisualizer	moblogger2	openlearningms	pvoice	sleaf	torneo	xmlrpc	lucene_nutch
babelchat	coushe	ecal	gjtapi	japm	jimps	jviz	mocl	openmetix	pvwikisystem	slgbd	touchbase	xmlsports	maven
babelfiche	cplab	ecdsainterface	glicompress	jarsync	jinx00	jvtelnet	modeclipse	openopac	pwts	slimdog	touchgraph	xmlswingrender	axis2
backulinu	cpmake	echo2gantt	glyphedit	jasa	jipsi	j-waste	model4lang	openorb	pyjdbc	slurred	trackplus	xmlvomiter	commons_axiom
banprodms	crackleback	eclipse	gml4j	jasen	jjbox	jwebviews	mofisp	openquasar	pyjtf	smallbee	transform-swf	xmsf	commons_neethi
barbecue	create-gedcom	eclipsewinamp	gmod	jasim	jkc	jwords	molevolve	openrct	pylibpcap	smartgallery	traxui	xnapster	commons_policy
barcode4j	cricketcage	ecolosim	gnap	jasmin	jlib-api	jwurfl	molybdenum	openspmnik	pypgp	smarttable	treedl	xpstorm	commons_tcpmon
basher	crispy	econf	gnucula	jasminegame	jlogger-tool	jxpg	monsterjournal	opentaps	pyrasun	sming	treesap	xqtav	commons_xmlschema
basicledger	crosstest	efp	gnuprologjava	jasmin-parser	jmdsync	jylog	moodytunes	open-tas	pythia	smtphandler	triot	xquetzal	jaxme
batino	cryptrl	eightball	gnurant	jasperdesign	jmedialibrary	kammkala	moonbounce	openticket	qa-distri	snaglepuss	tttn	xrate	juddi
bb-linkchecker	crypty	eje	gnurl	jasperintel	jmemorize	kawa	mosaique	openvod	qdparser	snmpproxy	tuneology	xsfa	muse
beanvalidator	cs401eve	ejpcc	gnu-tr	jasperpal	jmibbrowser	kchat	motojl	openwars	qlor	snmpz	tunepiano	x-smiles	pubscribe
beauty-hair-mng	cservapplet	ekspos	gnutsp	jatengine	jmicr	kennismanager	mousetracs	openwire	qms2	snortalertmon	tuplespace	xso	sandesha1
bediary	csfax	el4j	grafx	jato	jmoldraw	kernelanalyzer	movietheque	openworkbench	qndmp3	snurt	turbogps	xsserver	sandesha2
beeblebrox	csp	elseforif	graphlab	java2mysql	jmp3renamer	kifupdown	mozlinker	openxava	qom	solidsim	turquaz	xtapi	scout
bekaffe	csresources	elw	green	javaabc	jmplayer	kisgb	moztrans	oradump	q-scripting	somnifugi	tweakmaster2k	xtheater	soap
benchmarksql	css2xslfo	ematgine	gremlin	javabase64	jmscts	klassie	mp3cover	orbgate	queried	sonia	twlog	xul4j	wsif
beozip	csvjdbc	emu6502	gridkit	javacalc	jmsg	kmemorize	mp3db	orome	queryviewer	sonicmail	twoday	xvibs	wsrf
berserk	csvtosql	emubrow	groab	javacat	jmsn	knockknock	mp3info	osa-net	quickfix	sonogram	ucam	xwdb	wss4j
betoffice	ctbjava	epoint	guber	javacom	jmulti	kobjects	mp3-server	os-ohio	quizzer	sourcetapcrm	uic	yabay	xmlrpc
bfre	ctbwin32	eqemupack	guess	javacommons	jmutil	ktnef	mp3tag	osthreads	rabinhash	spacemapper	uima-framework	yajp	xmlbeans
bie-gpl	cupofjava	esig	guitartuner	javaconsole	jmxmonitor	kumasy	mpeg7audioenc	otwg	racess	space-wars	ujac	yakwa	xmlgraphics_commons
biff	cvsgrab	esperanto	gumbo	javadbshell	jnca	kurumix	mpxj	ouoss	raddoom	spagobi	ujgrader	yawn2	xmlgraphics_fop
binxml	cvsmail	essentialbudget	gusi	javaddlib	jnerve	kutttpech	mrcp4j	oxvip	ramble	spike-test-gen	ulmxmlview	zandiros	
bioclipse	cvsview	eternalrealms	gvf	javadiff	jnessuslib	lacl	mrss	p2p-radio	randservice	spin-chirps	unicoderewriter	zatreex	
bioera	cvw	eucatalogue	hamsam	javaemailserver	jnetstack	laoe	msjdbcproxy	p3s	rb-w32mod	spok	unischeduler	zaval0000	
biometricsdk	cweb	euler-solutions	hanzihelper	javaflightsim	jnettool	lazy8ledger	mtosi-ri	pa-bot	rcml	springie	uriqa4j	zebedee	
birmail	cyl-viewer	evilfoxe	haphazard	javahmo	jnex	lazy-x	multimethod-net	packet-cd	rcpapps	springside	ussrp	zeitline	
bitchburn	d20a	evocomp	hborn	javaidmef	jniosocket	lcalc	munchkin	palmdisplay	rcptutorial	sprite2d	utf-x	zetadb	
bjcruncher	d2r-map	evolab	hearsay	javaldap	jnm	ldbc	muscle	pam-cuecat	rcxtools	spwrapper	uuiet	zfotw	
blitztest	d3rose	e-volve	hecate	javamud	jnotes	ldplayer	muse	paperclips	rdqlplus	sqljep	vainstall	zgc	
blojsom	daffodilreplica	evolvo	helianto	javaplot	jnotify	lectcomm	mvip	parallel	reactionlab	sqlminus	vanatimer	zimpro	
bluemarine	dailypage	eworkbook	hermesjms	javaplugin	joepaint	ledos	mwscript	parfumball	readseq	sql-minus	varsha	ztemplates	
bmicalc	daimonin	excelutils	hibernate	javapsionlink	joeq	legacy2linux	mycore	paros	realfc	sqlplusplus	vboxj	zvtm	
bmpseq	daisy	excelxslreport	hibernatesample	javarcx	joesnmp	lfgapp	mysqlebk	particlereality	realtimecomm	sqlprofiler	venusview		
bnetutils	dal	exeqlib	hipergate	javascripttools	johnnyvon	libdict	myster	passkeyper	reason	sqooler	verdantium		
bnf-for-java	daoexamples	exhaust-us	hipo	javascriptzip	jopenphone	licensemanager	mytextreport	passwordkeeper	receipttracker	squirrel-sql	verto		
bocalendar	daomedge	expectj	hl7rdms	javasign	jopt	licq-forwarder	mythsim	patang	recognizer	srmccp	vexp		
boiler-plate	dataaccess	expense-ss	homermx	javasteps	jor	lilypad	myxdm	patmus	recurrance	sshwebproxy	vhstrain		
bonzo	datashare	expression4j	homerun	javatest	jorastat	limim	n-able	patsystem	redmond	sslexplorer	virtigex-comm		
bookdb	db2db	exspiminator	hotelcalifornia	javaunlimited	jorillas	lineageevolver	nathan	pcemu	regelectores	stakeout	virtualdb		
borg-calendar	db4obrowser	exymen	hotsheet	javujavu	josso	lipidia	nativecall	pedro	relaxed	startgater	virtualyou		
botp2004	dbappbuilder	eye	hpbtc	jazzlib	joti	littlisp	natura	peersim	remote2	state	visualdbscript		
bplusdotnet	dbatsol	eyesray	hrdag-analyzer	jbcd	journeysat	lix	navi-project	pennylender	repeatit	state4j	visualpostgres		
bpwj	dbconnection	facets	hsc-imsi	jbeginner	joutlookbar	lmdb	ncc	penny-pincher	replayer	stdnet	visualxmleditor		
bricksviewer	dbsql2xml	fairdj	htcommunicator	jbel	jpackit	lofimo	nemo2	penrose	reportregengine	stikiweb	vitapad		
brownbag	dbunit	fasper	html2pop3	jbilling	jpaul	logicalloy	neodesk	perl-md5-login	retina	strace	viva		
bscweasel	dbxml-core	fastrix	html2sql	jbmanageit	jpd	logitest	neonzip	persianmozilla	returnmypicture	strandz	vjdbc		
burnbabyburn	dcf	fate	hvc2003	jbotrace	jpfop	logshark	nepalivoice	personalaccess	reuse	strata	vmpsd		

Fig. 6. List of all 1555 projects used in assessing ranking schemes

[13] Anthony Cox, Charles Clarke, and Susan 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] Michael Eichberg, Mira Mezini, Klaus Ostermann, and Thorsten Schafer. Xirc: A kernel for cross-artifact information engineering in software development environments. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering (WCRE'04)*, pages 182–191, Washington, DC, USA, 2004. IEEE Computer Society.

[15] Joseph (Yossi) Gil and Itay 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.

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

[17] Reid Holmes and Gail 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.

[18] Katsuro Inoue, Reishi Yokomori, Hikaru Fujiwara, Tetsuo Yamamoto, Makoto Matsushita, and Shinji 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.

[19] Katsuro Inoue, Reishi Yokomori, Tetsuo Yamamoto, and Shinji Kusumoto. Ranking significance of software components based on use relations. *IEEE Transactions on Software Engineering*, 31(3):213–225, 2005.

[20] Rajeev Motwani Lawrence Page, Sergey Brin and Terry 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)*.

[21] Mark 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.

[22] David Mandelin, Lin Xu, Rastislav Bodík, and Doug 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.

[23] Andrian Marcus, Andrey Sergeyev, Vaclav Rajlich, and Jonathan 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, November 2004.

[24] Denys Poshyvanyk, Andrian Marcus, and Yubo Dong. Jiriss - an eclipse plug-in for source code exploration. *icpc*, 0:252–255, 2006.

[25] Diego Puppin and Fabrizio Silvestri. The social network of java classes. In Hisham Haddad, editor, *SAC*, pages 1409–1413. ACM, 2006.

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

[27] Naiyana Sahavechaphan and Kajal Claypool. Xsnippet: mining for sample code. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 413–430, New York, NY, USA, 2006. ACM Press.

[28] Susan Elliott Sim, Charles L. A. Clarke, and Richard C. Holt. Archetypal source code searches: A survey of software developers and maintainers. In *IWPC*, page 180, 1998.

[29] Renuka Sindhgatta. Using an information retrieval system to retrieve source code samples. In Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa, editors, *ICSE*, pages 905–908. ACM, 2006.

[30] Janice Singer, Timothy Lethbridge, Norman Vinson, and Nicolas 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.

[31] Kris De Volder. JQuery: A generic code browser with a declarative configuration language. In *PADL*, pages 88–102, 2006.