Search Engine for Open Source Code

Snigdha Rao Parvatneni

Sourcerer

Sourcerer

- Sourcerer is code search engine, designed to search open source code in Java (Bajracharya et al. 2006).
- Sourcerer enables user to search Java source code for–
  - Implementation of specific functionality at different granular levels e.g. implementation of a function, implementation of an entire component etc.
  - Uses of available piece of code at different granular levels e.g. uses of a function, uses of collection of functions as whole etc.
  - Code with specific patterns or properties e.g. code with concurrency constructs etc.

Components of Sourcerer

According to Bajracharya et al. (2006), Sourcerer has following components:

- **External Code Repositories** - internet based open source code repositories.
- **Local Code Repository** - repository for maintaining local copy of projects and meta data of each project.
- **Code Database** - relational database for storing features of the source code.
- **Code Crawlers** - different crawlers targeting different sources like well known repositories, arbitrary code at web servers etc.
- **Parser** - for parsing source code files present in local repository, extracting features, and storing them in code database.
- **Text Search Engine** - for searching with keywords.
- **Ranker** - for computing entity ranks using different ranking techniques.

Feature Extraction

- Entities, keywords, relations and fingerprints are collectively known as features. To extract these features parser works in multiple passes (Bajracharya et al. 2006).
- **Features -**
  - **Entities** – identified uniquely from source code by fully qualified name. List of entities extracted during parsing are – package, interface, class, method, constructor, field and initializer.
  - **Relations** – identified as dependency between two entities. List of relations are extends, implements, calls, throws, returns, inside, instantiates, read, writes, overrides, overloads and uses.
  - **Keywords –** identified as meaningful text associated with entities. Keywords are extracted from fully qualified names and comments.
  - **Fingerprints –** quantifiable features associated with entities. Fingerprints are numbers implemented as d dimensional vectors. It allows to compute similarity using methods like cosine distance etc.

Feature Storage

- **Relational Database** (Bajracharya et al. 2006)
  - It uses source model which consists of two tables to store program entities and their relations respectively.
  - For each entity compact representation of attributes are also stored for fast retrieval like fingerprints and indexed keywords.
  - In an additional pass through database source-to-source inter project links are resolved.
- **Fingerprints** (Bajracharya et al. 2006)
  - Fingerprints are used to support structure based searches of source code.
  - Three types of fingerprints used in Sourcerer are –
    - **Control Structure Fingerprints** – It captures control structure of an entity like number of loops number of conditionals etc.
    - **Type Fingerprints** – It captures information about type like number of implemented interfaces, number of declared methods etc.
    - **Micro Patterns Fingerprints** – It captures information about implementation patterns like class with no fields etc.

Search Application

According to (Bajracharya et al. 2006)

- General purpose infrastructure is used for indexing source code.
- Keywords entered by user is matched against the set of keywords maintained by Lucene.
- Each keyword is mapped to list of entities and each entity has its rank and other information like version, location in source code etc. associated with it.
- On querying list of entities matched against the sets of matching keywords can be returned as search results to the user.

Ranking Search Entities

According to (Bajracharya et al. 2006), ordering search results using some measure of relevancy.

- **Heuristic 1: Packages, Classes and Methods Only (Text based method)**
  - Implementation of functionality are assumed to be wrapped in classes or in methods. So, while searching for implementations, focus in names of classes, methods, and packages and ignore everything else.
- **Heuristic 2: Specificity (Structure based method)**
  - Take advantage of ordered containment relationships among packages, classes and methods to rank the search results. E.g. hits on method name is more valuable than hits on class name and hits on class name is more valuable than hits on package name.
- **Heuristic 3: Popularity (Graph based method)**
  - In Sourcerer code rank model components are represented as weighted directed graph where nodes represent classes, methods, fields etc. and edges represent cross component usage. Graph rank algorithm is used to compute component rank. All the JDK classes are ignored and no pre processing of graph is done to cluster similar code.

Assessment and Results

According to Bajracharya et al. (2006),

- Standards used for assessing Sourcerer are Recall and Precision. Precision is kept fixed and Recall is given more importance.

- Criteria used to select good hits from result list are – contents of the result, quality of the result in terms of completeness and reputation of the project from which the solution is selected.

- Comparison of text based methods, structure based methods, and graph based methods and their combinations, for more relevant hits on the first page, tells that combination of all three methods will return better search results.

Google Code Search Engine

Introduction

According to Cox (2012),

- Google code search engine works well with regular expression queries.

- This code search feature was developed on top of already existing document indexing and retrieval tools.

- It is very difficult to keep all the source code files in memory to perform search action. That’s why Code search user inverted index to identify candidate documents where snippet can be searched.

- Matching document are then scored and ranked to be shown as search results.

Indexed Regular Expression Search

- When inverted index is based on words then regular expression searches does not lines up correctly in word boundaries. To eliminate this problem we build index based on character n grams (Cox , 2012).

- Value for n is chosen as three because for two there are very few distinct words and for four there are too many distinct words. That’s why trigrams are used to build inverted index (Cox , 2012).

Example

This example is described in Cox (2012),

“

(1) Google Code Search

(2) Google Code Project Hosting

(3) Google Web Search

Here 1, 2 and 3 represents document numbers.

There corresponding trigram index is –

_Co: {1, 2}	Sea: {1, 3}	e_W: {3}	ogl: {1, 2, 3}
_Ho: {2}	Web: {3}	ear: {1, 3}	oje: {2}
_Pr: {2}	arc: {1, 3}	eb_: {3}	oog: {1, 2, 3}
_Se: {1, 3}	b_S: {3}	ect: {2}	ost: {2}
_We: {3}	ct_: {2}	gle: {1, 2, 3}	rch: {1, 3}
Cod: {1, 2}	de_: {1, 2}	ing: {2}	roj: {2}
Goo: {1, 2, 3}	e_C: {1, 2}	jec: {2}	sti: {2}
Hos: {2}	e_P: {2}	le_: {1, 2, 3}	t_H: {2}
Pro: {2}	e_S: {1}	ode: {1, 1}	tin: {2}

Here _ represents space

”

Query For Candidate Documents

This example has been taken from Cox (2012)

- “Query to search regular expression like /Google.*Search/ would be -
  Goo AND oog AND ogl AND gle AND Sea AND ear AND arc AND rch”

- According to Cox (2012), we can use this query over previously mentioned trigram index to find candidate documents and then search full expression over these candidate documents.

Rules To Compute Regular Expression

This explanation has been taken from Cox (2012)

- **“ ‘’ (empty string)**
  - emptyable(‘’) = true
  - exact(‘’) = {‘’}
  - prefix(‘’) = {‘’}
  - suffix(‘’) = {‘’}
  - match(‘’) = ANY (special query: match all documents)

- **c (single character)**
  - emptyable(c) = false
  - exact(c) = {c}
  - prefix(c) = {c}
  - suffix(c) = {c}
  - match(c) = ANY ”

From Cox (2012)

- “***e*****? (zero or one)**
  - emptyable(*e*?) = true
  - exact(*e*?) = exact(*e*) ∪ {‘’}
  - prefix(*e*?) = {‘’}
  - suffix(*e*?) = {‘’}
  - match(*e*?) = ANY

- ***e*** *** (zero or more)**
  - emptyable(*e**) = true
  - exact(*e**) = unknown
  - prefix(*e**) = {‘’}
  - suffix(*e**) = {‘’}
  - match(*e**) = ANY

- ***e*****+ (one or more)**
  - emptyable(*e*+) = emptyable(*e*)
  - exact(*e*+) = unknown
  - prefix(*e*+) = prefix(*e*)
  - suffix(*e*+) = suffix(*e*)
  - match(*e*+) = match(*e*) ”

From Cox(2012)

- “ **e_1 | e_2 (alternation)**
  - emptyable(e_1 | e_2) = emptyable(e_1) or emptyable(e_2)
  - exact(e_1 | e_2) = exact(e_1) ∪ exact(e_2)
  - prefix(e_1 | e_2) = prefix(e_1) ∪ prefix(e_2)
  - suffix(e_1 | e_2) = suffix(e_1) ∪ suffix(e_2)
  - match(e_1 | e_2) = match(e_1) OR match(e_2)

- **e_1 e_2 (concatenation)**
  - emptyable(e_1e_2) = emptyable(e_1) and emptyable(e_2)
  - exact(e_1e_2) = exact(e_1) × exact(e_2), if both are known or unknown, otherwise
  - prefix(e_1e_2) = exact(e_1) × prefix(e_2), if exact(e_1) is known or
    prefix(e_1) ∪ prefix(e_2), if emptyable(e_1) or
    prefix(e_1), otherwise
  - suffix(e_1e_2) = suffix(e_1) × exact(e_2), if exact(e_2) is known or
    suffix(e_2) ∪ suffix(e_1), if emptyable(e_2) or
    suffix(e_2), otherwise
  - match(e_1e_2) = match(e_1) AND match(e_2) ”

Simplification

- Rules described in last section does not produce good quality queries and if we use those queries then result set will be unmanageable. To keep result manageable we can apply some simplification rules at each step. Below mentioned example has been taken from Cox (2012)
- Example for single string (Cox, 2012)
  - “trigrams(mn) = ANY
  - trigrams(mnp) = mnp
  - trigrams(mnpq) = mnp AND npq”

- Example for set of strings (Cox, 2012)
  - “trigrams({mn}) = trigrams(mn) = ANY
  - trigrams({mnpq}) = trigrams(mnpq) = mnp AND npq
  - trigrams({mn, mnpq}) = trigrams(mn) OR trigrams(mnpq)
    = ANY OR (mnp AND npq) = ANY
  - trigrams({mnpq, stuv}) = trigrams(mnpq) OR trigrams(stuv)
    = (mnp AND npq) OR (stu AND tuv)”

Transformation

- Transformations can be applied to any regular expression during analysis. Transformations conserves validity of the results and are applied to keep results manageable. Below mentioned example has been taken Cox(2012)

- For saving information (Cox, 2012)
  - "At any time, set match(*e*) = match(*e*) AND trigrams(prefix(*e*)).
  - At any time, set match(*e*) = match(*e*) AND trigrams(suffix(*e*)).
  - At any time, set match(*e*) = match(*e*) AND trigrams(exact(*e*)). "

- For discarding information (Cox, 2012)
  - "If prefix(*e*) contains *m* and *n* and *m* is a prefix of *n* then discard *n*.
  - If suffix(*e*) contains *m* and *n* and *m* is a suffix of *n* then discard *n*.
  - If prefix(*e*) is very large then remove the last character off the longest strings in prefix(*e*).
  - If suffix(*e*) is very large then remove the first character off the longest strings in suffix(*e*).
  - If exact(*e*) is very large then set exact(*e*) as unknown."

- Best way to use this is to use information saving before discarding the information (Cox, 2012).

References

[1] Bajracharya, S., Baldi, P., Dou, Y., Linstead, E., Lopes, C. Ngo, T., & Rigor, P. (2006, October). *Sourcerer: A search engine for open source code*. Paper presented at Proceedings of the 2006 Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, Oregon, Portland, USA. Retrieved on February 12, 2013 from http://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=3&sqi=2&ved=0CEMQFjAC&url=http%3A%2F%2Fwww.researchgate.net%2Fpublication%2F228804354_Sourcerer_A_Search_Engine_for_Open_Source_Code%2Ffile%2F79e4150930483860a6.pdf&ei=VWaQUfSMEeOQjAKgwIGAAg&usg=AFQjCNHLSP_h9lrEkiJysTyVdZhLlCczgg&bvm=bv.46340616,d.cGE

[2] Cox, R. (2012, January). *Regular Expression Matching with a Trigram Index*. Retrieved on February 15, 2013, from http://swtch.com/~rsc/regexp/regexp4.html