A SOURCE CODE SEARCH ENGINE FOR KEYWORD BASED STRUCTURAL RELATIONSHIP SEARCH

by

ASHEQ HAMID

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT ARLINGTON
May 2013

To my parents who had always been my main source of courage, enthusiasm and support.

ACKNOWLEDGEMENTS

There are so many people to thank to. First of all, I would like to express my thankfulness to my supervisor Dr. Christoph Csallner and Dr. Chengkai Li for their constant help and support. Without their assistance, I would have never been able to complete the thesis. They have shown me the direction all through the way, encouraged me when I was down and appreciated me to put in more effort. Also they have helped me with their immense knowledge in the field to overcome any problem when I was stuck. I would like to thank Dr. Jeff Yu Lei for taking out his time from busy schedule for being in my advising committee and sharing his comments on my work.

I would like express my gratitude to Sarker Tanveer Ahmed Rumee for his active participation in this research work and for his knowledge and contribution for it. Also I am grateful to all the members of Software Engineering Research Center, specially Ishtiaque Hussain and Mainul Islam for making the place an enjoyable place to work.

I would like to appreciate the support of the Department of Student Enrollment Project Management Group for providing me assistantship while continuing my research. Specially, I would like to express my gratitude to my supervisor Dennis O'Hare and director Robert Gunnip for allowing me a flexible schedule when it was necessary. I would also like to share my gratefulness to my current employer EBSCO Publishing for allowing me to work for them and at the same time to continue my research work. I would like to thank my manager Dhirendra Sharma and director Ken Lafrance for being so understanding to me.

I owe my gratitude to my mother Hosne Ara Hamid who had always been a source of my courage. Also I would like to remember my late father Dr.A.K.M Abdul Hamid whose dream was to see me going for this accomplishment. My special thanks go to my loving wife Naima Islam for being by my side through my tough times. I would also like to express my humble gratefullness to my siblings Aminara Hamid and Tareq Hamid for being so supportive to me.

In the end, I would like to express my gratefulness to Allah whose blessings had always been with me.

March 26, 2013

ABSTRACT

A SOURCE CODE SEARCH ENGINE FOR KEYWORD BASED STRUCTURAL RELATIONSHIP SEARCH

ASHEQ HAMID, M.S.

The University of Texas at Arlington, 2013

Supervising Professor: Christoph Csallner

In an Object Oriented Program, we often see that a package contains several classes, a class contains several methods, a method calls other methods. We may say, there is a *contains* relationship between a package and a class or a *calls* relationship between two methods. We refer to these relationships as *structural relationships*. There may be other structural relationships apart from *contains* or *calls* in the source code. A software developer may sometime want to search for structural relationships within source code. She may prefer using Google like free form query to do so. To facilitate free form query based structural relationship search in the source code, we have proposed a code search system. Our system allows a user to type free form query and find structural relationships from the code repository relevant to that query. We consider an Entity Relationship Model where class, method etc. are entities and structural relations between entities are the actual relations. We construct a directed data graph taking each entity element as a node and each relation between elements as an edge. Each node in the data graph has a name (such as *foo*) and a type (such as *method*). Each edge has a type (such as *calls*). For each node, we construct a neighborhood document that lists the names and types of the nodes and types of the edges within edge length d from that node along with their respective distance score. The closer is the node or edge from the given node, the higher is the distance score. We use these neighborhood documents to construct inverted index and then do a Google like search. We find the nodes in the data graph where all the query keywords can be found in the neighborhood and order the nodes based on a simple ranking mechanism. To verify the effectiveness of our proposed code search model, we have implemented a prototype considering one structural relation. We have come up with some possible user queries and evaluated our system for each of those queries. Our experiment findings show that, our system is fairly successful in finding out the related structural relationships the user may be looking for.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

CHAPTER 1

INTRODUCTION

A code project written in an object oriented programming language is often full of many relationships. For example, a project may contain several packages, each package may contain multiple classes, each class may have multiple methods, each method may have many local variables and so on. These are examples of "contains" relationship. A method may call some other method in its body. That is an example of a "calls" relationship. We can mention many such relationships due to the structure of object oriented programming. We call these structural relationships.

Sometimes a programmer may feel the need to search for some structural relationship in the source code. In that case, she may have to use an existing code search engine or she may use grep over some code documents. Although there are existing search engines which allow searching for such structural relationships, the user must need to provide a search query according to the syntax that engine understands. However, the searching could have been much easier if she could express her search intention as a free form query. By free form query, we mean that a user may type whatever comes to her mind to express her query. For example: A user may want to type foo calls bar or foo invokes bar to find a method foo that calls a method bar. So we feel, it may be useful if there is a code search engine that enables structure based relationship search from the source code through free form query. Hence in this thesis, we have proposed the model of such a code search engine. At the same time we have implemented a prototype of the system to see how our proposed model works in reality.

1.1 Motivation

Source code search is a fairly common task done by software developers. Although it is fairly common, relatively little is known about how and why developers perform code search [1]. Not too many surveys have been done on this particular topic. A useful study have been published by Sim et al. [2]. According to them, code search is often done during the development and maintenance of software. They mentioned several reasons for searching code. The main four motivations have been mentioned as (1) defect repair, (2) code reuse, (3) program understanding, and (4) impact analysis. Some example scenarios for code search may be: A developer is informed about a defect in a software project, she may search within that project to find the reason of the defect. When a developer is trying to understand the flow of a program, she may search within that project to find references to a method or a variable. On the other hand, impact analysis is done when some change is made to a codebase. Impact analysis ensures, if the new change has broken any other part of the code. So a developer may be interested to search all the references from different part of that project to the particular method or variable which she has modified. Searching for code reuse can be further explained by a couple of motivating examples.

Example 1: A developer may be implementing a new feature in a software using an API. She may look into the API documentation and find out a method to accomplish a task. The API documentation may not have example code of how to use the method. So she may search for code that calls that API method. From the search result she may get the examples of the usage of the method. She can get the idea about how to use that method and then implement herself. She may also just copy the code if that exactly matches her needs.

Example 2: A developer may be looking for a sort method that returns an integer array. So she may search for such a method within other code projects. If she gets such a method then she may just reuse that method.

According to [3] a user's search for reusable code can be at different levels of granularity. For example, a user may search for a collection of methods that does some task or a single method that performs something. However, regardless the granularity level of the search, it may be useful to find a number of relevant code examples from different projects when a developer is searching for reusable code. That is why one of our motivations is to develop a source code search engine that is not limited to a single project but is capable of searching across multiple projects.

It may not be ideal to treat source code documents just like 'bags of words' [4]. This means that it may not be ideal to treat the words in a code document as independent occurrences without any code context. We may ignore the structural relationships among the words in a code document if we just treat the document as some collection of words. Unfortunately, ignoring the relations among words may lead to reduced search accuracy. Source code is full of structural relationships among the code components such as classes, methods, variables etc. It may be necessary to take these structural relationships into account for the advent of next-generation code search technologies [5]. This motivates us to propose a code search engine that allows structural relationship search among the code components such as package, class, method etc..

It is fairly important to choose the type of user input query model the proposed system should support. A user of the system should feel comfortable specifying her query. Now a days, Google-like free form query is quite popular. For a user, it is easy to use since she can type and express what she wants to search in plain English. She does not have to learn any new query syntax or any new query language. Also it makes the input interface really simple. If it is a web based interface, then a textbox may be good enough to take user query. On the other hand, it may be difficult for the system to interpret the user's exact search intent from the query. As the query grows bigger, it becomes more complex to identify the user's intent. Sometimes, a user intent and the system interpretation for the query may not match. So the results may be quite different from what she expects. Although there are difficulties to support free form query, still its simplicity to use is a strong motivation for us to choose a keyword based free form query interface.

There are existing search engines that implement keyword based structural relationship search. Sourcerer [3] is the most relevant search engine on this regard. If there is a relation name such as "calls", "contains" etc. in the input query, this system may not interprete that the user may be looking for these structural relationships in particular. We realize that, if a user can specify the relation name as part of the query and a search engine can consider that relationship while searching for relevant answers, that may help finding answers closer to user's intent. This may reduce the percentage of non relevant answers. So we propose a novel search technique that attemps to take into account structural relations that are specified in the user query along with other program component names. The search results that are returned by our system reflects those relationships among program components. For example: if a user types "method foo calls method bar", then our system tries to find a method named "foo" which "calls" a method named "bar". This novel technique makes our system different from any existing work.

1.2 Challenges

There are several tasks associated with our code search system. Roughly, we can break down the major tasks into the following:

1. Collecting source code projects from third-party repositories.
2. Extracting the data required from the code base.
3. Searching from that data by free form query.
4. Presenting the result via code snippets.

Our main focus of this thesis is on item no. 3 mentioned above. We have tried to address the challenges related to this item. For the challenges under remaining tasks, we either do little work just to support the work under item no. 3 or fully leave that out of the scope of our current work. In the following, we discuss in greater detail some of the major challenges.

Download source code: Collecting code projects from different repositories require different approaches. We are interested in collecting Java projects as our code search model is proposed for Java. We prefer to collect the bytecode as well as the source code for a project. The bytecode can be used for data extraction and the source code can be used for displaying code snippet. We have done little work to deal with these challenges. We have written a program that downloads code projects from the Apache software repository. We follow a heuristic to collect both bytecode and source code for a downloaded project. We have left detailed work to address these challenges out of the scope of this thesis.

Data extraction from the code: We want to do static analysis of the code and find different code entities such as classes and methods. from the code. We choose to analyze Java bytecode rather than the actual source code. There are a couple of reasons for that. Firstly, Java bytecode is an intermediate form generated after compilation of the source code. The program construct in the bytecode is comparatively simpler than in the Java source code. Hence it is easier to analyze bytecode compared to source code. Secondly, there are existing well-known tools/libraries which can analyze Java bytecode. We may use a suitable tool/library to extract the data we need. One such Java bytecode analysis tool is Soot [6]. Soot is a Java bytecode analysis tool and soot-fact-generator is a part of it. This fact-generator provides us many of our desired entities and relationships by analysing the bytecode. However, the problem of using soot-fact-generator is:

1. We may not get all types of entities/relationships we may be looking for.
2. It may be difficult to interpret soot-fact-generator's output at times which makes it difficult to use for our work.
3. This tool may not be suitable for future expansion.

ASM [7] is a static analysis library for Java bytecode engineering. We find ASM good enough for our work. That is why we have written our data extraction module using the ASM library. This enables us to extract the data we need from the code. Also it keeps the opportunity wide open to extract other types of information that we have not yet considered for the current work. We find the library suitable for future enhancement of our work.

Finding a fast search technique based on free form query: A user may write anything in a free form query. Some keywords in the query may be more important than the others. Again the positioning of the keyword in the query may carry special significance. For example: "foo calls bar" and "bar calls foo" both have the same keywords but due to the position of the keywords, the meaning may be different. It is quite difficult for the system to understand the user intent from the free form query. It is a big challenge to interpret the free form query and quickly find out the results that are most relevant to the query keywords. To address this type of problem, some research work as in [8, 9] used a tuple based approach to directly search in a relational database. [10] used a form-based approach combined with keyword search to address similar issue. [11] proposes a keyword based structured query language to search on web extracted data. To cope with this problem, we devise a datagraph based algorithm that finds out code entities relevant to the free form query. Our algorithm also gives importance on keywords which indicate relation names. We have introduced indexing in our search technique which results in faster retrieval of the search result. For the current scope of our work, we have not considered Natural language processing (NLP) for the parsing of the free form query to better understand the user intent.

Display the output: Our search engine performs structural relationship search. Due to this nature of the search, the output display is different from traditional full-text search engines. In full-text search engines, the portions of a document which contain matched terms from keyword query are usually displayed. For example, if a user query is ”foo bar”, then the result may show the documents that contains the keywords ”foo” or ”bar”. Structure based code search may require displaying code snippets that may be far apart in the same document or may consist of multiple documents. For example : A query ”method calls run calls execute” may return a method ”m1” which is in document Doc1. In Doc1, ”m1” may call a ”run” method, but it is possible that the ”run” method calls the ”execute” method in document Doc2.

Displaying the output in a suitable manner is a challenging problem and require some mechanism to solve this. We have kept this problem out of the scope of our work.

1.3 Contribution of the Thesis

In this thesis, we propose a technique that allows searching for structural relationship from source code by free form query. Our system takes into account any structural relationship's name mentioned in the free form query and tries to find answers which exhibit structural relationships of that kind. For example, if a user query says ”foo calls bar”, then our system gives importance on the ”calls” keyword and tries to find answers where keywords ”foo” and ”bar” are related through the ”calls” relationship. We consider source code in an Entity Relationship model. We represent the code entities and structural relationships as a data graph. Our technique shows a way to map each node and the nearby keywords of that node as a document. We associate each keyword with some score. The closer the keyword from the given node, the higher is it's score in the document. We index those documents and conduct a Google-like search on those documents to find the desired answers. We implement a prototype of our system and also show that our technique is fairly successful at finding structural relationships that are mentioned in the query.

1.4 Organization of the Thesis

The organization of the thesis is as follows. The initial part of Chapter 2 discusses the background work. The later part of the chapter defines some related concepts and states the research problem. Chapter 3 proposes the solution to the problem. It describes the necessary definitions, techniques and algorithms for covering details of the proposed solution. Chapter 4 discusses implementation details. The experiments are described in Chapter 5. This chapter also evaluates the outcome of those experiments. The conclusion and the possible future expansions are mentioned in Chapter 6.

CHAPTER 2

BACKGROUND AND PROBLEM STATEMENT

In this chapter, we give an overview of the previous work that has been done on code search. After that, we define some basic concepts that may be helpful for understanding our problem domain. In the end, we define the problem we have worked on.

2.1 Related Work

Over the years, several code search engines have been made to support different code search needs. They differ on various aspects such as underlying code search technology, types of input supported, types of output produced etc. Traditional code search engines are the ones that usually do full-text search on the source code documents. Chromium [1], Krugle [2] and Koders [3] are examples of such engines. These engines also allow searching for limited structural information. Chromium and Koders allow searching for method name and class name. Krugle can search for method definition and class definition along with name. However, these search engines are not ideal for structural information search. We mention a couple of reasons for that. Firstly, these engines usually allow searching for a predefined limited set of types, namely class, method etc. There are many other types such as array, enum, constructor etc. that cannot be directly searched. Secondly, relationships among these types can hardly be searched. For example, we may search for class name but

[1]http://code.google.com/p/chromium/source/search
[2]http://opensearch.krugle.org/
[3]http://www.koders.com/

we may not search for subclass-superclass relationship. So there remain limitations that gave further scopes to improve source code search.

Codase [4] and Merobase [5] are two search engines with more advanced structure based search along with full-text search. Both engines added method search based on method signature and class search based on class type. Additionally, Codase can distinguish among class field declaration, field reference and a variable. Although these engines have shown slightly broader capability for structural search, specifying a particular structural information in the search query is still difficult. The developer needs to use the query syntax for that specific engine to write a query. This criterion may not be desired for the developer. There are other advanced languages like .QL [12] and JQuery [13] for searching fine grained structural information from the source code. .QL allows an OOP programming model to find complex structural relationships. JQuery uses the Abstract Syntax Tree to find such information from the code. The system in [14] uses a visual query editor to easily form complex queries to find code based on structural relationship from the source code. However, developers may not be interested to learn a new language or even a new query syntax for searching the source code. Although learning the syntax for some of those engines might be easier than learning a new language, still this learning factor may discourage the user from using that search engine.

In order to help users to avoid learning a new language or a new syntax, some code search engines support searching for structural relationships using a free from query. Sourcerer [3] is a source code search engine which has created an infrastructure for structural search on source code collected from open source software repositories. Their keyword based search engine allows free form queries. Although this engine

[4]http://www.codase.com/

[5]http://merobase.com/#main

has structural relationship search capability, specifying the relation's name as part of the free form query is not possible. While searching, the user may have to look into each of the returned search results and find out the one that actually shows the relationship she is looking for. There is a possibility that none of the returned results would contain the relationship she is looking for. This makes it difficult for the user to find an exact structural relationship quickly and easily. Other related works are the tool in [15] and strathcona [16]. [15] is a system that uses keyword query along with class type, method signature, test case, contract, security constraint to find a relevant code snippet. Strathcona [16] uses structural context from the code written by the user to retrieve relevant code example. To get better output, a user needs to write a code snippet in a convention suitable for understanding of strathcona. This may not be desired by the developer.

Several other code search tools have concentrated on finding limited number of structural informations to satisfy needs for the corresponding research context. [4] is an improvement to conventional text based code search to find more relevant method examples. It considers the location of the keyword match whether it is in the method name or in the method body, semantic role of the matched word, frequency of occurrence of the matched keyword in the document etc. on top of the text-based search result to find relevant code examples. SE-CodeSearch [17] is a tool which uses Abstract Syntax Tree based approach to find a few structural relations from an uncompilable piece of code. Their main focus is on finding information from an uncompilable piece of code rather than supporting a greater number of relations. Portfolio [18] is a code search tool which accepts free form query and finds a relevant call graph. Codegenie [19] uses test cases to search it's repository to find an implementation for the method for which the test case may be a good fit. PARSEWeb [20] can find code example given a source object and a target object. The code snippet will consist of a method sequence that will be able to generate the target object starting with the source object. Other relevant approaches may be useful while looking for particular use of an API or to find examples to accomplish a task. [21] and [22] are free form form query based tools which find relevant API usage examples.

None of the aforementioned work exactly matches our approach. Our system has an easy to use free form query interface to relieve the user from learning any new query language or syntax. Also our system allows specifying structural relation's name as part of the query and it takes the relation name into account while searching for relevant answers.

2.2 Definitions and Data Model

In the next few subsections, we define some basic concepts. Also we define our data model alongside those concepts.

2.2.1 Set

A set is a collection of elements. We define each of the following as a set.

- Package
- Type - interface, class, enum, nested type of all modifiers (protected, final, static, etc.)
- Method - constructor, static initializer of all modifiers (native, abstract, private, static, etc.)
- CodeContainer - Union of Package, Type, and Method.
- Operator - We only consider instanceof operator.
- Word

We do not consider the following important enough for the kinds of high-level code understanding tasks we envision for the search engine. That is, we do not need the following:

- Field
- Local Variable
- Operator: It may be tempting to add operators, but program operators could easily be confused with query operators (!, &&, &, etc.)
- Modifier such as:
    - public, private, protected
    - static
    - abstract
    - final
    - strictfp, transient, volatile
    - synchronized
    - native
- Type being a class, interface, or enum
- Type parameters such as Foo in List$< Foo >$
- Import
- Annotation
- Control-flow such as
    - if, else, switch, case
    - for, while, do, break, continue
- throw, try, catch, finally
- assert
- return

Example: Let us consider the following code example. We'll find sets from this example. We will use the same code example for later definitions as well.

```
package edu.uta.cse.td;
/*@author csallner@uta.edu (Christoph Csallner)*/
public class TipCalc
{
  /*Calculate tip.*/
  public static void main(final String[] args)
  {
    if (args==null || args.length != 2)
    {
     System.out.println("need 2 parameters");
     return;
    }
    final int a = Integer.parseInt(args[0]);
    final int b = Integer.parseInt(args[1]);
    final int res = (a * b) / 100;
    System.out.println(res);
  }
}
```

The following example sets use fully qualified names (e.g., "edu.uta.cse.td.TipCalc" instead of the simple name "TipCalc"). This is done here to be able to distinguish between elements that may have the same simple names; But this does not mean that the implementation must also use fully qualified names.

Package = { edu, edu.uta, edu.uta.cse, edu.uta.cse.td, java, java.lang, java.io }

Type = { void, int, edu.uta.cse.td.TipCalc, java.lang.Integer, java.lang.System, java.io.PrintStream }

Method = {

void edu.uta.cse.td.TipCalc.main(java.lang.String[]),

void edu.uta.cse.td.TipCalc.$< init >$(),

int java.lang.Integer.parseInt(java.lang.String),

void java.io.PrintStream.println(int),

void java.io.PrintStream.println(java.lang.String)

}

Word = { csallner@uta.edu, Christoph, Csallner, Calculate, tip }

2.2.2 Relation

Given sets A and B, a relation between A and B is a set of ordered pairs (a, b) such that a $\in$ A and b $\in$ B . Each ordered pair is called a tuple. A relation can be among multiple sets. When the relation is between two sets, then we call it a binary relation. Each ordered pair (a,b) is called a binary tuple. Given the sets defined before, we define the following binary relations among those sets. The left side of ':' is the name of the relation and the right side is the name of the two sets between which the binary relation exists.

- contains_pp: (Package, Package)
- contains_pt: (Package, Type)
- contains_tt: (Type, Type)
- contains_tm: (Type, Method)
- is_subtype: (Type, Type)
- overrides: (Method, Method)
- returns: (Method, Type)

- takes: (Method, Type)
- throws: (Method, Type) - declared in signature or thrown explicitly
- calls: (Method, Method)
- instanceof: (Method, Type) - method has expression ”x instanceof Type”
- describes: (Word, CodeContainer)

In this model, there is either zero or one (but not more) ”calls” relationship between any method m1 and method m2. We think, the number of times a call to method ”bar” appears in a ”foo” method body is not a reliable indicator of how often ”foo” calls ”bar”. For example: we can consider the case where ”foo” has a loop that calls ”bar” or ”foo” itself is invoked in a loop.

Example: Following are the relations that we get from the aforementioned code example:

contains_pp = {

(edu, edu.uta),

(edu.uta, edu.uta.cse),

(edu.uta.cse, edu.uta.cse.td),

(java, java.lang),

(java, java.io)

}

contains_pt = {

(edu.uta.cse.td, edu.uta.cse.td.TipCalc),

(java.lang, java.lang.Integer),

(java.lang, java.lang.System),

(java.io, java.io.PrintStream)

}

contains_tm = {

(edu.uta.cse.td.TipCalc, void edu.uta.cse.td.TipCalc.main(java.lang.String[])),

(edu.uta.cse.td.TipCalc, void edu.uta.cse.td.TipCalc.TipCalc()),

(java.lang.Integer, int java.lang.Integer.parseInt(java.lang.String)),

(java.io.PrintStream, void java.io.PrintStream.println(int))

(java.io.PrintStream, void java.io.PrintStream.println(java.lang.String))

}

is_subtype = {

(edu.uta.cse.td.TipCalc, java.lang.Object)

}

returns = {

(void edu.uta.cse.td.TipCalc.main(java.lang.String[]), void),

(void edu.uta.cse.td.TipCalc.$< init >$(), void),

(int java.lang.Integer.parseInt(java.lang.String), int),

(void java.io.PrintStream.println(int), void),

(void java.io.PrintStream.println(java.lang.String), void)

}

takes = {

(void edu.uta.cse.td.TipCalc.main(java.lang.String[]), java.lang.String[]),

(void edu.uta.cse.td.TipCalc.$< init >$(), void)

}

calls = {

(void edu.uta.cse.td.TipCalc.main(java.lang.String[]),

int java.lang.Integer.parseInt(java.lang.String)),

(void edu.uta.cse.td.TipCalc.main(java.lang.String[]),

void java.io.PrintStream.println(int)),

(void edu.uta.cse.td.TipCalc.main(java.lang.String[]), void java.io.PrintStream.println(java.lang.String)),

}

describes = {

(csallner@uta.edu, edu.uta.cse.td.TipCalc),

(Christoph, edu.uta.cse.td.TipCalc),

(Csallner, edu.uta.cse.td.TipCalc),

(Calculate, void edu.uta.cse.td.TipCalc.main(java.lang.String[])),

(tip, void edu.uta.cse.td.TipCalc.main(java.lang.String[])),

}

2.2.3 Synonyms of set and relation names

Function: Given two sets A and B, a function F from A to B is a set of ordered pairs (a, b) such that for some a $\in$ A, there is exactly one b $\in$ B. The function F is called a total function if, for each and every element in A there is a mapped element in B. Otherwise, the function is called a partial function.

We define a function that maps synonyms to our set and relation names. Some example tuples of the function are:

- (Interface,Type)
- (Class,Type)
- (Enum,Type)
- (Type,Type)
- (Code, Method)
- (Block, Method)
- (Constructor, Method)
- (Method, Method)
- (implements, is_subtype)

- (extends, is_subtype)
- (is_subtype, is_subtype)
- (invoke, calls)
- (invokes, calls)
- (calls, calls)

2.2.4 Data Graph

Given our previously defined sets and relations,

- We assume each element of each set is a vertex. A vertex has a ID and a type. The ID is the fully qualified name of the element and the type is the set name in which the element belongs to. For example, ”edu.uta.cse.td.TipCalc.main” is a ID and ”method” is a type. We define a vertex set V_d consisting of all the vertices.
- We assume, for each tuple of each relation there is a directed edge from the vertex representing first element to the vertex representing second element of that tuple. Each edge has a type which is the relation name in which the tuple belongs to. For example, for the tuple (Christoph, edu.uta.cse.td.TipCalc), we assume a directed edge from vertex representing element ”Christoph” to vertex representing element ”edu.uta.cse.td.TipCalc”. ”describes” is the type name for that directed edge. We define an edge set E_d consisting of all the edges.

Based on the above assumptions, we define a directed graph G_d with vertex set V_d and edge set E_d. This graph $G_d(V_d,E_d)$ is our data graph.

Example: We create the datagraph shown in Fig. 2.1 from the derived sets and relationships of the above example.

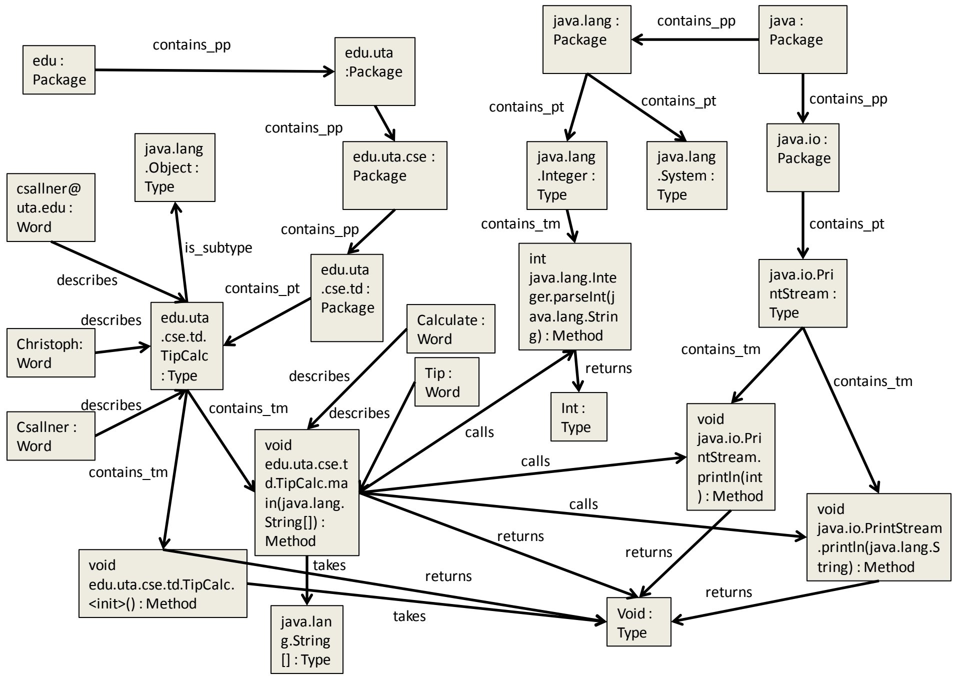

Figure 2.1. Data Graph.

2.2.5 Neighborhood of a node

Each vertex in the datagraph is also called a node. For a node N in the datagraph, all the other nodes (including N) and edges within distance D of N construct the neighborhood of that node. Here, by distance we mean shortest distance in terms of edge length from N. The distance of N itself is 0. The distance of a node N1 one edge away from N is 1. The distance of an edge E1 adjucent to N is 1. The distance of an edge adjucent to N1 is 2.

Example: We consider the node java:Package from the datagraph in Fig. 2.1. The neighborhood of this node will consist of the following sets of vertices and edges when distance D=2. The distance for each element is mentioned within first brackets.
Vertices = {
java.package(d=0), java.lang:Package(d=1) , java.io:Package(d=1),

java.lang.Integer:type(d=2), java.lang.System:type(d=2),

java.io.PrintStream:type(d=2)

}

Edges = {contains_pp(d=1),contains_pt(d=2)}

2.2.6 Free Form Query

A free form query is a query where a user can write anything to express her question. There is no specific syntax for the query. The same search intention may be expressed in numerous ways. For example, each of the following free form queries may be used to find a method named "foo" that calls another method named "bar".

1. Method foo calls method bar.
2. Find a method foo that calls foo.
3. foo calls bar
4. foo invokes bar
5. bar called by foo

2.2.7 Stop words for queries

Given a free form query Q, stop words are a list of words that should be filtered out before answering the query. These words either carry little meaning or we do not yet support them. Some examples of stop words are : a, the, find, search, that, not, and, or, implies, if, then, iff etc.

2.3 Problem Statement

Since we have already defined the basic concepts, now we want to define the actual research problem. We divide our problem statement into the following parts:

1. Our input to the system is a free form query Q. Given a free form query Q, we want to filter out the stop words from Q and find a set of keywords K = {k_1, k_2,k_3,...,k_n} to use for finding the answer.
2. Given a set of keywords K = {k_1, k_2,k_3,...,k_n} and a datagraph $G_d(V_d,E_d)$, a node N_A in the data graph is our answer node if, for each keyword $k_i \in K$ atleast one of the following is true:
   (a) k_i is an ID of a node N1 $\in V_d$ and N1 is in the neighborhood of N_A.
   (b) k_i is a type name of a node N2 $\in V_d$ and N2 is in the neighborhood of N_A.
   (c) k_i is a type name of an edge E1 $\in E_d$ and E1 is in the neighborhood of N_A.

   There may be multiple nodes in the datagraph that may satisfy one of the above conditions. So there may be multiple answer nodes.
3. Given a set of answer nodes, we want to order the answers according to some ranking mechanism.

What we describe above is our short term goal. We try to achieve this goal in this thesis. However, the long term goal is to find out how the keywords in the neighborhood of an answer node are connected to each other through some stuctural relationships. We also want to make suitable code snippets and display to the user.

CHAPTER 3

PROPOSED SOLUTION

To address the problem described in section 2.3, we have proposed a code search system. This system collects code projects, extracts sets/relations from the code, constructs data graph and uses our proposed technique to find the answer nodes from the data graph. In the following sections we discuss about the architecture of the system and the internal details of it's different modules.

3.1 Architecture of the system

The overall system consists of three basic modules.

1. Code Collector Module
2. Data Extractor Module
3. Search Module

In Fig. 3.1, we show different modules and how they are related to each other. Code collector module fills up local repository collecting code from external repository. Data extractor module analyses the code from the local repository and fills up database with extracted data. Search module takes user query, database data as input and produces answer nodes. In the following subsections, we look into each of these modules in greater detail.

3.1.1 Code Collector Module

Our target is to search within code projects collected from online repositories. The job of this module is to collect code projects from online repositories. For this

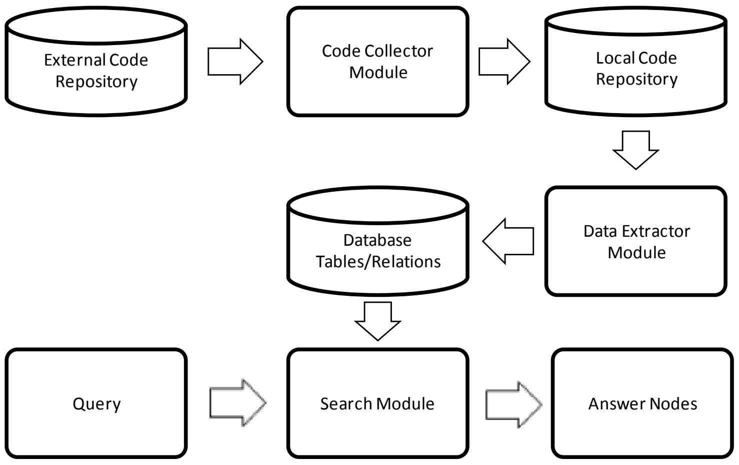

Figure 3.1. Architecture of the code search system.

thesis, we have downloaded code projects from Apache software repository. Following are some details of the actions of this module for collecting code projects from Apache software repository:

1. Sometimes several zipped formats of a project may be available. In that case, we may not download duplicates for the same version of a project. For example: apache-abdera-1.1.2-src.zip and apache-abdera-1.1.2-src.tar.gz are two different zipped formats for the same project. We want to download one of those.
2. We may get only source content, only bytecode content, both source and bytecode content from the downloaded project folder. For a bytecode file, we want to locate the corresponding source file for the purpose of code snippet display. If the downloaded project folder contains both the bytecode and the source then we may need to locate which bytecode file corresponds to which source file. Sometimes there are separate downloadable folders for bytecode and source code. For this thesis work, we follow a heuristic approach to download both

bytecode and source code of a project. After downloading, we manually find out the bytecode - source code pair.

3. After detecting the bytecode file and the corresponding source file, this module stores those files to the local repository.

3.1.2 Data Extractor Module

On a high level, the task of this module can be divided into the following:

1. Static analysis of the bytecode files from the local repository: By static analysis [1] we mean analysis of bytecode without actually executing the code. We do static analysis on the the bytecode files from our local repository. We do static analysis because, the type of data we are interested in, may be extracted by doing static analysis of the bytecode. We do not need dynamic analysis for the current scope of our work.
2. Extract data from the bytecode content: The type of data we want to extract from the bytecode has been described in the data model. We have used ASM [7], a bytecode engineering library for this purpose. ASM provides an API that can go over the bytecode and allows us to extract several structural information such as class name, method names of a class, method starting/ending line number etc.. We have found, ASM has the support to extract the type of information we may want to consider. We have written a tool that does the data extraction using the ASM bytecode engineering library. Our tool requires multiple passes over the bytecode to extract the data we need for current implementation. The details design and implementation of this tool has been done my Sarker Tanveer Ahmed Rumee.

[1]http://en.wikipedia.org/wiki/Static_program_analysis

3. Create the database tables and relations according to schema: We execute a SQL script, that creates the database tables for entities and relations. The schema of the tables that we have considered for this work has been shown in Table 4.1
4. Populate database tables and relations with extracted data: Our tool extracts the data from bytecode and then populates the database tables.

3.1.3 Search Module

This module takes user query and database data as input. We already know that the database data is a collection of code entities and relations between those entities. This module uses an algorithm to find the code entities that may be related to user query. These code entities are the potential answers. Since there may be multiple candidates for the answer, this module uses a ranking scheme to order those candidates. In the next section we describe our search model in detail.

3.2 Search Model

In the search model, we first introduce some of the basic concepts we have used for the model. Then we describe our query model. In the end, we describe the search algorithm along with the ranking mechanism.

3.2.1 Basic concepts

Special characters: We consider the following as special characters:

1. '_', '0', ... '9' are often used as part of a method/type identifier. For example: A method name may be "do_work()".
2. '.' is often used as a delimiter in package.type.method.

   For example: "edu.uta.cse.td.TipCalc.main"

3. ’(’, ’)’ are often used to specify a method parameter list.

   For example: ”myMethod(myInteger,myString)”.
4. ’[’, ’]’ are used to denote an array type. For example: ”java.lang.String[]”

Free variables: We may consider an example scenario. A user may want to search for all methods that call method ”foo”. We could use a star (*) operator for that. The user may say ”* calls foo”. However, maybe more elegant is allowing the user to use not only the name of relations but also the name of sets. Some example user queries may be:

1. method calls foo
2. class extends List
3. class extends List and contains a method that calls bar
4. foo calls a method that calls bar

Stop words for the data graph: We want to use a fixed list of words that should not appear in the data graph, as they carry little meaning. These words should only come from comments (Word set). Some examples of such words are:

– a, the, we, that, this

– first, next, finally, and, or, then, if, iff etc.

Simple name and Qualified name: Simple name does not contain a ’.’; Some examples of simple names could be:

– fooPackage

– FooType

– FooType[]

– fooMethod

– fooMethod()

Qualified name contains at least one ’.’; Some examples of qualified names could be:

– fooPackage.FooType

– fooPackage.FooType.fooMethod

– FooType.fooMethod

The user may do an initial search with a simple name such as ”println”. This may return results for both ”PrintStream.println” and ”Foo.println”. To focus the search on a certain element, the user may later refine this part of the query, by replacing the simple name ”println” with a qualified name such as ”PrintStream.println” or ”java.io.PrintStream.println”.

3.2.2 Query Model

In this subsection, we define our query model using EBNF [2] form.

$\langle \mathit{query} \rangle ::= \langle \mathit{keyword} \rangle$

$\quad | \quad \langle \mathit{query} \rangle\ \langle \mathit{blank} \rangle{+}\ \langle \mathit{keyword} \rangle;$

$\langle \mathit{blank} \rangle ::= \quad ;$

$\langle \mathit{keyword} \rangle ::= \langle \mathit{identifier} \rangle$

$\quad | \quad \langle \mathit{identifier} \rangle\ \langle \mathit{blank} \rangle{*}\ \texttt{(}\ \langle \mathit{blank} \rangle{*}\ \texttt{)}$

$\quad | \quad \langle \mathit{identifier} \rangle\ \langle \mathit{blank} \rangle{*}\ \texttt{(}\ \langle \mathit{blank} \rangle{*}\ \langle \mathit{params} \rangle\ \langle \mathit{blank} \rangle{*}\ \texttt{)}$

$\quad | \quad \langle \mathit{identifier} \rangle\ \langle \mathit{arraySign} \rangle;$

$\langle \mathit{arraySign} \rangle ::= \texttt{[]}$

$\quad | \quad \langle \mathit{arraySign} \rangle\ \texttt{[]};$

$\langle \mathit{params} \rangle ::= \langle \mathit{identifier} \rangle$

$\quad | \quad \langle \mathit{identifier} \rangle\ \langle \mathit{arraySign} \rangle$

$\quad | \quad \langle \mathit{params} \rangle\ \langle \mathit{blank} \rangle{*}\ \texttt{,}\ \langle \mathit{blank} \rangle{*}\ \langle \mathit{identifier} \rangle$

$\quad | \quad \langle \mathit{params} \rangle\ \langle \mathit{blank} \rangle{*}\ \texttt{,}\ \langle \mathit{blank} \rangle{*}\ \langle \mathit{identifier} \rangle\ \langle \mathit{arraySign} \rangle;$

[2] http://en.wikipedia.org/wiki/Extended_Backus-Naur_Form

$\langle \textit{identifier} \rangle$ ::= $\langle \textit{simpleName} \rangle$

| $\langle \textit{identifier} \rangle$ `.` $\langle \textit{simpleName} \rangle$;

$\langle \textit{simpleName} \rangle$ ::= $\langle \textit{char} \rangle$+;

$\langle \textit{char} \rangle$ ::= `_`, `0`, ... `9`, `a`, .., `z`, `A`, .., `Z`;

3.2.3 Search Technique

Now based on the previously defined basic concepts, data model and query model, we describe a technique to find the answers from the user query. There are multiple steps associated to our proposed technique. In the remaining part of this chapter, we describe each of those steps and the associated algorithms.

3.2.3.1 Overview of Search Technique

The basic idea is to treat the node names and node/edge types that are close by a given node in the data graph as a document, build an inverted index, and do a Google-like search.

3.2.3.2 Name/Type vs. ID

In the data graph, each node represents a code element and has both a name (for example: "foo") and a type (for example: "method"). Each edge has a type (for example: "calls").

For each code element and relation we distinguish among it's name and type and it's id. The id is unique. The name is a simple name such as "foo" and may not be unique.

By default, a code element or relation in the following refers to an id. If we mean a name of a code element or relation, we write Name(x). Similarly, we denote the type of an element with Type(x).

3.2.3.3 Per-Node Document N.hood of Close-By Node/Edge Names/Types

These documents are mappings of nodes to keywords.

Node $\rightarrow$ Keywords

Node $\rightarrow$ Node names / Node types / Edge types

For each node N in the data graph we create one document named "N.hood". (We do not create any document for edges). A document lists the names and types of the nodes and types of edges in N's neighborhood and their respective distance score from N. The closer the node or edge to N, the higher its score. This is similar to word frequency in a classic document. There the more often a keyword appears, the higher its score or frequency.

We will use these neighborhood files to construct the inverse index files described in section 3.2.3.4.

Specifically, a document has the following content expressed in EBNF:

$\langle \textit{document} \rangle$ :: = ($\langle \textit{entry} \rangle$ \n)*;

$\langle \textit{entry} \rangle$ :: = (Name(node), score)

| (Type(node), score)

| (Type(edge), score);

3.2.3.4 Per-Name and Per-Type Index Name.idx / Type.idx of Nodes that have Name/Type Close-By

These documents are mappings of keywords to datagraph nodes :

Keyword $\rightarrow$ Nodes

For our problem scenario, node names/types and edge types are matched with query keywords. In other words, these documents are mapping of:

Node name $\rightarrow$ Nodes

Node type $\rightarrow$ Nodes

Edge type $\rightarrow$ Nodes

For each name/type that is used by a node or edge, we create an index file. An index file lists the data graph nodes in which proximity the name/type is used by a node.

We will use these index files to quickly find a node that has a neighborhood that includes all names/types (keywords) of a given query.

Specifically, an index file has the following content expressed in EBNF:

$\langle index \rangle$ :: = ($\langle entry \rangle$ \n)*;

$\langle entry \rangle$ :: = (node, score);

3.2.3.5 Construction of N.hood Documents

We propose Algorithm 1 to construct the N.hood documents. This algorithm is a modified Depth First Search. It takes the datagraph and a maximum depth D_max as input. Then it applies DFS on each node of the datagraph digging maximum D_max depth from a node. The outcome of the algorithm is a set of N.hood documents having one document for each node in the datagraph.

Algorithm 1 Neighborhood file generation algorithm

Input: Data graph

Output: A set of neighborhood files.

```
1: for all node N in the data graph do
2:     Create document N.hood for N
3:     Add to N.hood: (Name(N), D_max)
4:     Add to N.hood: (Type(N), D_max)
5:     Perform a depth-first search (DFS) on the data graph,
       initial node is N, up to depth D_max,
       DFS interprets each edge as an undirected edge
6:     for all edge E = (N_A, N_B) traversed by DFS at depth D,
       distance(N,N_A) = D-1, distance(N,N_B) = D, distance(N,E) = D do
7:         score(D) := (D_max - D)
8:         if N.hood does not contain Name(N_B) then
9:             Add to N.hood: (Name(N_B), score(D))
10:        end if
11:        if N.hood does not contain Type(N_B) then
12:            Add to N.hood: (Type(N_B), score(D))
13:        end if
14:        if N.hood does not contain Type(E) then
15:            Add to N.hood: (Type(E), score(D))
16:        end if
17:    end for
18: end for
```

3.2.3.6 Construction of Inverted Index

Once we get all the N.hood documents, we construct the inverted index files from those. The purpose is to find all the nodes related to a keyword quickly. We propose Algorithm 3 for creating the inverted index files. The input of the algorithm is a set of N.hood documents. The output is a set of inverted index files.

3.2.3.7 Search Algorithm

In this section, we describe Algorithm 2 which is used for searching. This algorithm takes a set of keywords and all the index files as input. The output is a set of result nodes ordered by a ranking scheme. The algorithm returns only those nodes for which all the input keywords are found in close proximity of that node. The ranking scheme is very simple and has been described as part of the algorithm. The closer the input keywords are from a resulting node, the higher is the rank for that node.

3.2.3.8 Re-construction of query graph from result nodes

For displaying the code snippet corresponding to the result nodes, we need to have some mechanism. We propose two solutions for displaying code snippet. The first one is a simple one. The second one is an elegent solution. We have not implemented the second one and kept it out of the scope of our work for this thesis.

Basic Solution: We load and display result node code entity. We want to let the user browse the code and locate the keywords, they are close-by.

Nicer Solution: We need to enhance the index documents by adding to each entry the respective neighborhood node defined in 3.2.3.3 and 3.2.3.5. This allows us to load the document N.hood for each result node. Then for each keyword in the

Algorithm 2 Search algorithm

```
Input: A set of keywords, index files
Output: A set of data graph nodes
1: For each keyword k,
   load the corresponding keyword.idx files,
   k_1.idx, .., k_W.idx
2: Find R result nodes N_1, .., N_R
   each result node N is listed in every k_i.idx keyword file,
   $1 \leq i \leq W$
3: for all result node N do
4:     Compute score(N),
       score(N) = score(N in k_1.idx) + .. + score(N in k_W.idx)
5: end for
6: Rank result nodes by their scores
```

query, we need to look up the node of that name in N.hood. This gives us all the nodes that correspond to the query keyword. Then we need to find out a mechanism to create suitable snippets of these nodes and show them to the user.

Algorithm 3 Index file generation algorithm

```
Input: A set of neighborhood files
Output: A set of index files.
 1: for all document N.hood do
 2:     for all line do
 3:         Append N
            {this yields rows of forms:
            (Name(node), score, node)
            (Type(node), score, node)
            (Type(edge), score, node)}
 4:     end for
 5: end for
 6: Merge all rows from all documents into a single file
 7: Sort rows by the first column (name/type)
 8: for all value in first column do
 9:     Extract rows with that value into file value.idx
10:     Delete first column
11:     Swap remaining columns
12:     Order rows by score
13: end for
```

CHAPTER 4

IMPLEMENTATION

We have implemented a quick prototype of our system choosing one structural relation from the relations we have discussed in section 2.2.2. We have chosen ”calls” relation to implement our system and to evaluate the results. In this chapter, we discuss how we have implemented the system. We talk about implementation of all 3 modules of the system one by one. In each phase we have faced some challenges and we have solved those challenges taking different approaches. In the rest of this chapter, we focus on these implementation details.

4.1 Implementation of code downloader module:

We have written a program in Java to download projects from apache code repository for our search engine implementation. We provide the root URL for an apache code repository to the program. The program uses Java URL class to download code projects in our local system. The organization of project folders in apache repository is fairly straight forward. The structure is like a directory tree. There is a main folder and under the main folder, there are several other sub folders. Each sub folder corresponds to an individual apache project. For each project folder, there may be subfolders under that. Sometimes, there are subfolders for different released versions of that project. At the very bottom level of the directory tree lies the actual project archived as some compressed file. zip, jar , tar.gz are some common compression file types that we can mention. Our main idea is to traverse the directory tree from the root to the very bottom level until we get the compressed project file. Then we want to download the project file from that level. Since there is no concrete directory structure followed for all the projects or there is no concrete naming convention for the folders, we have faced some challenges to download projects. Sometimes we have used heuristics to solve some of them. We discuss about those implementation issues in the following subsections.

4.1.1 Depth of traversal

To avoid digging very deep into the folders and subfolders of the repository, we used a depth d from the root. If we get the archived project file within that depth d, we download that file. If we do not get the archived project file within depth d, then we quit digging deeper. For our implementation, we have assigned d=5.

4.1.2 Heuristic to decide availability of a compressed project file at a given depth

We download the page content from the given root URL. We parse all the hyperlinks from that page. Then we dig into each of the hyperlinks parsed from the page. For a hyperlink, we follow a heuristic to decide if we want to dig further deep into it or not. The heuristic is: we look for any .jar/.zip/.tar.gz extension under a URL. If we find any, then we assume that the hyperlink contains the compressed project file. We decide not to dig any further. If we do not get any compressed file, we assume that we need to dig deeper .

4.1.3 Selecting compressed file type

We prefer downloading .jar file for a project. If we get .jar file for the project we download it. If we do not find .jar file then we look for .zip and .tar.gz extensions respectively. We do not consider any other compressed file format for this implementation.

4.1.4 Heuristic for selecting latest project version

Sometimes there are folders for multiple release versions of the same project. For apache repository, the version number is often part of the project folder name. If a folder contains multiple subfolders containing different versions of the project, we parse the version numbers from the subfolder names. We calculate a numeric value for each version number. By calculating numeric value , we mean, if a version number is 1.0.0.50, the numeric value for the version is 10050. We assume, the version number having the greatest numeric value is the latest one. We decide to dig deeper into that latest release folder of the project.

4.1.5 Heuristic for selecting projects with source and bytecode

In apache repository, if a compressed file contains source code for the project, then the file contains the term ”src” or ”source” in its file name. We look for one of these terms in the compressed filename. If we find the presence of one of these terms, we assume, the file contains source code. We look for the term ”bin” or ”binary” in the compressed file name to decide if the file contains only bytecode. We download a project only when we get both the binary and the source for that project.

4.2 Implementation of data extraction module

There are two parts associated with this module. First part is creating database tables. We have used MySQL as the database. Second part is populating the database tables with data. This module uses an application written in Java to extract data from Java bytecode and populate the tables. This module has been designed and implemented by Sarker Tanveer Ahmed Rumee. We think it may be helpful to have some insights into this module. In the following subsections we focus on some details of this module.

Table 4.1. Schema for database tables

Table name	Column names
package	(id,name)
class	(id, packageid, name, containerFile)
method	(id, classid, name, returntype, startlinenumber, endlinenumber)
caller_callee	(caller_id, callee_id, linenumber)

4.2.1 Creation of entity and relation tables in the database

This module creates the entity tables that are required for supporting calls relation. These tables are package, class and method. In addition to these entity tables, this module creates one relation table caller_callee. The schema for these tables are given in Table 4.1.

4.2.2 Data extraction from the code projects and population of the database tables

This module uses a bytecode engineering library, ASM version: 3.3.1 for performing this task. ASM provides an API to analyze Java bytecode and extract structure based information from it. ASM is itself written in Java. This module uses this library to go over class files from our downloaded codebase and extract information and populate the database tables. It requires two passes over the class files to extract all the information we need for our current implementation.

4.3 Implementation of search module

We have written an application in Java to perform the search. This application has three independent parts. Each of these parts produces output independent of other parts. We discuss about these parts in the following:

4.3.1 Neighborhood files generator

This part communicates with the database and creates a data graph. Then it applies the neighborhood file generation algorithm (discussed in section 3.2.3.5) on the data graph. The output is a number of neighborhood files one for each node in the data graph.

4.3.2 Index generator

This part works on the neighborhood files and does not depend on the data graph for its functionality. We have implemented index generation algorithm (discussed in section 3.2.3.6) in this part. The output of this part is a set of index files.

4.3.3 Implementation of search algorithm

This is the final part which uses the index files for searching and does not depend on neighborhood files or data graph. We take user input and parse the keywords from it. For the initial prototype, we have not given much stress on the parser implementation. We find the keywords by splitting the user query on white space. We have implemented the search algorithm (discussed in section 3.2.3.7) in this part. The output is a set of answer nodes. For display, we fetch starting line number of the answer entity and show 30 lines of code beginning from the starting line. For example, if a method named ”execute” is returned as an answer, we show 30 lines of code that starts from the beginning of the method definition upto 30 following lines.

CHAPTER 5

EXPERIMENT AND EVALUATION

We conduct some experiments with the system we have developed. The goal for this experiment is to evaluate how our proposed search technique performs. We come up with some example queries that a user may come up with. We give those queries as input to the system and save the output. Then we manually scrutiny each of those results and find out whether they are accurate in our context or not. In the following sections, we discuss more about our experiments.

5.1 Experiment setup

There are multiple tasks associated with experiment setup. In the following subsections we discuss about those tasks.

5.1.1 Project selection

We randomly choose 8 projects from the set of projects, we have downloaded using our code downloader module. These 8 apache projects are listed in Table 5.1

Table 5.1. List of apache projects

No.	Project name	No.	Project name
1	abdera	5	cassandra
2	ant	6	camel
3	aries	7	cayenne
4	avro	8	xmlbeans

Table 5.2. Number of rows in database tables

Table name	Number of rows
package	1788
class	1929
method	21580
caller_callee	7002

We manually separate the .class files and the corresponding .java files and put them in the designated folder in our local repository.

5.1.2 Table setup and data population

We write a sql script that creates the database tables. We manually run that script in MySQL database to create the tables we need. Then we run the data extraction tool on the chosen 8 projects. This has been able to extract data from those projects and polulate the database tables. After the data population is complete, our database is populated with the number of rows as shown in Table 5.2.

5.1.3 Neighborhood file generation

We run the neighborhood file generation algorithm and create the neighborhood files. We have noticed that 25297 files have been generated. This number is the expected result, since one neighborhood file is created corresponding to each data graph node. From the number of database rows, we can see that, we have 1788 rows in package table, 1929 rows in class table, 21580 rows in method table. Each row corresponds to one node in the data graph. So we should have (1788 + 21580 + 7002 = 25397) nodes in the data graph. This refers to 25297 neighborhood files which is the actual number of generated .hood files.

5.1.4 Index file generation

We run the index generation algorithm on 25397 .hood files. We have seen that 10449 index files are generated.

5.2 Experiment and results

Since the experiment setup is complete, now we conduct the actual experiment. We come up with some example queries. We implement only "calls" relation for the initial prototype of the system. So the queries we come up with are all related to finding "calls" relation in the code base.

To select the method names to be used in the query we look into our database data. The reason for looking into database data is to come up with some method names for which good number of output should be generated. Following are the queries we come up with:

1. execute calls run
2. method calls decode
3. method calls getClient
4. toJson calls method calls writeField
5. poll calls processBatch calls isBatchAllowed
6. execute calls checkOptions calls getReplace
7. getRecordWriter run calls create
8. getRecordWriter run calls method
9. checkOptions calls getComment getCommentFile
10. method calls release schedule
11. method calls method
12. method calls

We pass these example queries as inputs to our system. For each of these queries, a number of results are generated. Each result is a node in the data graph. To be more specific, in our case, each result is a method id that corresponds to a row in method table. For experiment's sake, we have chosen to evaluate the correctness of top 5, 10, 15, 20 and 25 results. We try to come up with some practical queries that a real user may put to the system. For example: We do not try any example like "foo calls x calls y calls bar" with more than one "calls" relation, since we think it will hardly be the case that a user may be interested to write such a query.

5.3 Evaluation

Before we evaluate the results, we would mention a couple of concepts that we have used to evaluate our results.

Query Intent: For each of the experiment queries mentioned before, we fix a query intent. We evaluate our results against the fixed intent. Following list shows the query intents for each query.

1. execute calls run : Find an example where a method named execute calls a method named run.
2. method calls decode : Find all the methods that calls a method named decode.
3. method calls getClient : Find all the methods that calls a method named get-client.
4. toJson calls method calls writeField : Find a method that calls another method named writeField and is called by a method named toJson.
5. poll calls processBatch calls isBatchAllowed : Find an example where a method named polls calls another method named processBatch and processBatch calls a method named isBatchAllowed.

6. execute calls checkOptions calls getReplace : Find an example where a method named execute calls another method named checkOptions and checkOptions calls a method named getReplace.
7. getRecordWriter run calls create : Find an example where a method named getRecordWriter and a method named run both calls a third method named create.
8. getRecordWriter run calls method : Find an example where a method named getRecordWriter and a method named run both calls a third method.
9. checkOptions calls getComment getCommentFile : Find an example where a method named checkOptions calls two other methods; one is named getComment and the other is named getCommentFile.
10. method calls release schedule : Find an example where a method calls two other methods; one is named release and the other is named schedule.
11. method calls method : Find all the methods that calls some other method.
12. method calls : Find all the methods that calls some other method.

Evaluation of accurate answers: For a user query, our system returns some datagraph nodes. For a returned datagraph node, we look into it's neighborhood to see if the entities and relations expressed by the query intent truly exist in the neighborhood. If they exist then the answer can be considered accurate. Otherwise we consider the answer as inaccurate. One example of an inaccurate answer may be: We assume, a user provides a query ”execute calls run”. Her query intent is to find a method ”execute” that calls a method ”run”. The system returns ”run” or ”execute” as an answer node, and the corresponding neighborhood contains a directed edge from ”run” to ”execute”, representing that ”run calls execute”. This does not go with the query intent. So we consider the answer as inaccurate.

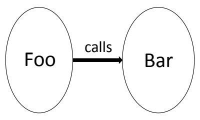

Figure 5.1. Two method nodes related through a calls relation.

Duplicate answers: There may be duplicate answers returned by our system. We explain duplicates with an example. Let us assume the user query is "foo calls bar". The query intent is to find an example where a method named "foo" calls a method named "bar". This is shown in figure 5.1. Since our system returns datagraph nodes, it may be possible that both the nodes corresponding to "foo" and "bar" are returned. As we'll look into the neighborhood of both nodes, we'll find the same entities/relationships. Although our system may return two different nodes, but actually both of them refer to the same answer. Hence, there may be duplicates among the answer nodes by our system.

Precision and Recall: Precision and recall are two common concepts for evaluating an information retrieval system. Precision is the number of relevant results a search retrieves divided by the total number of results retrieved, while recall is the number of relevant results retrieved divided by the total number of existing relevant results that should have been retrieved. In other words, let us assume that the ground truth has N1 answers for a user query. The system returns N2 answers, and the size of N1 $\cap$ N2 is N3. Then we can define precision and recall as:

$$precision = N3/N2$$

$$recall = N3/N1$$

5.3.1 Evaluating precision and recall for our system

Evaluate precision: We calculate precision at k where the value of k may be 5, 10, 15, 20, 25. For precision at k, we consider top k nodes returned by our system. We know that our system may return duplicate answers that is multiple answer nodes may refer to the same answer. So we calculate precision at k both with duplicates and after removing duplicates. To calculate precision at k with duplicates (P_{kd}), we find the number of accurate answers (AA_{kd}) among top k. Here we consider the nodes that refer to the same answer as separate individual answers. Then we divide the number of accurate answers by k. We can formulate the following folmula for calculating precision at k with duplicates:

$$P_{kd} = \frac{AA_{kd}}{k}$$

To calculate precision at k after removing duplicates (P_{kwd}), we find unique answers among top k (UA_k). This means, if there are multiple nodes that refer to the same answer, then we consider those nodes under one unique answer. These unique answers may contain both accurate and inaccurate answers. Then we find the accurate unique answers from the unique answers (AUA_k). We divide accurate unique answers by unique answers to calculate precision at k after removing duplicates. We can formulate the following folmula for calculating precision at k after removing duplicates:

$$P_{kwd} = \frac{AUA_k}{UA_k}$$

Evaluate recall: To evaluate the recall of the system, we need to find all the correct answers possible for a query. For a user query, we exhaustively check our database data to find all occurances that goes with the query intent. The number of occurances we get, we consider that as the total number of accurate answers for that query. In other words, this number is the ground truth for that query (GT).

We calculate recall at k where the value of k may be 5, 10, 15, 20, 25. Whatever, the value of k is, ground truth is always the same. We calculate recall at k only after removing dulicate answers. For calculating recall at k after removing duplicates (R_{kwd}), we find unique accurate answers among top k (AUA_k). This means, if there are multiple nodes that refer to the same answer, then we consider those nodes as one unique answer. Then we divide the number of unique accurate answers by the number of ground truth. We can formulate the following formula for calculating recall at k after removing duplicates:

$$R_{kwd} = \frac{AUA_k}{GT}$$

We do not calculate recall at k with duplicates because that may result in a recall over 100% and may not be an interesting measure.

5.3.2 Evaluation of result

Since we have discussed how we want to calculate precision and recall for our system, now we present, what we have actually found after calculating precision and recall. Table 5.3 to table 5.7 show our experiment findings after evaluating top 5, 10, 15, 20, 25 answer nodes returned by our system along with calculated precision at k with duplicates and after removing duplicates. We also show recall at k after removing duplicates. We do not include Query 11 and Query 12 with this analysis. These two queries are very general and it may be difficult to manually find duplicates from the returned answers. Since out datagraph only contains "calls" relations, we assume each returned answer node for these two queries should have a "calls" relation within it's neighborhood. That is why, we consider each returned answer as accurate. We have only calculated recall with duplicates (R_d) for these two queries considering all the answer nodes. The result has been shown in table 5.8.

Table 5.3. Experiment findings for top 5 answer nodes; k=5

Query	GT	AA_{5d}	UA_5	AUA_5	R_{5wd}	P_{5wd}	P_{5d}
1	4	5	3	3	3/4≡75%	3/3≡100%	5/5≡100%
2	6	4	5	4	4/6≡66.66%	4/5≡80%	4/5≡80%
3	4	4	4	3	3/4≡75%	3/4≡75%	4/5≡80%
4	6	5	5	5	5/6≡83.33%	5/5≡100%	5/5≡100%
5	5	5	5	5	5/5≡100%	5/5≡100%	5/5≡100%
6	4	5	4	4	4/4≡100%	4/4≡100%	5/5≡100%
7	6	5	5	5	5/6≡83.33%	5/5≡100%	5/5≡100%
8	24	5	5	5	5/24≡20.83%	5/5≡100%	5/5≡100%
9	9	5	5	5	5/9≡55.55%	5/5≡100%	5/5≡100%
10	4	5	4	4	4/4≡100%	4/4≡100%	5/5≡100%

As we can see that the recall after removing duplicates (R_{kwd}) increases a bit as we consider larger values for k. One reason for this may be due to the presence of duplicate answers, our system may not return all the unique accurate answers when we consider a smaller value for k. As the value of k increases, we get more unique accurate answers. On the other hand, the precision with duplicates (P_{kd}) and after removing duplicates (P_{kwd}) may seem to decrease as we consider larger k values. One reason for this may be, as we consider more answer nodes, we tend to discover more inaccurate answers. We see 100% precision and recall values for some cases. This may not be an indication that our system is perfect in returning accurate answers most of the time. One reason for this may be, our database may not contain data to produce inaccurate answers for those queries. If we conduct our experiement on a larger amount of data considering more than one structural relationship, we may discover more inaccurate answers and the precision and recall may go down from 100%. While manually verifying each answer returned by our system, we came across

Table 5.4. Experiment findings for top 10 answer nodes; k=10

Query	GT	AA_{10d}	UA_{10}	AUA_{10}	R_{10wd}	P_{10wd}	P_{10d}
1	4	8	6	4	4/4≡100%	4/6≡66.66%	8/10≡80%
2	6	8	6	5	5/6≡83.33%	5/6≡83.33%	8/10≡80%
3	4	8	6	4	4/4≡100%	4/6≡66.67%	8/10≡80%
4	6	10	6	6	6/6≡100%	6/6≡100%	10/10≡100%
5	5	10	5	5	5/5≡100%	5/5≡100%	10/10≡100%
6	4	10	4	4	4/4≡100%	4/4≡100%	10/10≡100%
7	6	10	5	5	5/6≡83.33%	5/5≡100%	10/10≡100%
8	24	10	10	10	10/24≡41.67%	10/10≡100%	10/10≡100%
9	9	10	9	9	9/9≡100%	9/9≡100%	10/10≡100%
10	4	10	4	4	4/4≡100%	4/4≡100%	10/10≡100%

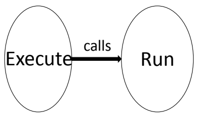

Figure 5.2. Data graph node showing execute method calls run method.

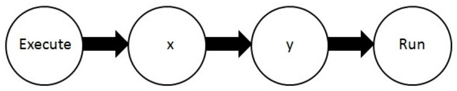

Figure 5.3. Data graph node showing execute method calls run method through two intermediate nodes.

some patterns for inaccurate answers. In the following we mention some of those with examples.

Example 1: We assume that a user is looking for a method "execute" that calls "run" as in fig 5.2. Our system sometimes return a method named "execute" because that method calls a method "foo" and "foo" calls "run". This in fact implies "execute" calls "run", but that is not what the user is looking for. So we have

Table 5.5. Experiment findings for top 15 answer nodes; k=15

Query	GT	AA_{15d}	UA_{15}	AUA_{15}	R_{15wd}	P_{15wd}	P_{15d}
1	4	9	7	4	4/4≡100%	4/7≡57.14%	9/15≡60%
2	6	11	9	6	6/6≡100%	6/9≡66.67%	11/15≡73.33%
3	4	11	6	4	4/4≡100%	4/6≡66.67%	11/15≡73.33%
4	6	15	6	6	6/6≡100%	6/6≡100%	15/15≡100%
5	5	15	5	5	5/5≡100%	5/5≡100%	15/15≡100%
6	4	15	4	4	4/4≡100%	4/4≡100%	15/15≡100%
7	6	12	6	5	5/6≡83.33%	5/6≡83.33%	12/15≡80%
8	24	14	14	13	13/24≡54.16%	13/14≡92.85%	14/15≡93.33%
9	9	15	9	9	9/9≡100%	9/9≡100%	15/15≡100%
10	4	15	4	4	4/4≡100%	4/4≡100%	15/15≡100%

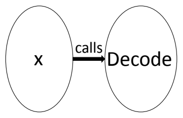

Figure 5.4. Datagraph node showing x method calls decode method.

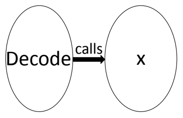

Figure 5.5. Datagraph node showing decode method calls x method.

considered such answers as incorrect. Also we have seen cases where "execute" calls "run" through two intermediate nodes as shown in fig 5.3

Example 2: Another example of inaccurate answer can be, a user is looking for a method that calls another method named "decode" as shown in fig 5.4. Our system may return a method that is called by a method named "decode" as shown in fig 5.5.

Example 3: The query intent is to find an example where both "getRecord-Writer" and "run" methods call "create" method. Our system returns an answer

Table 5.6. Experiment findings for top 20 answer nodes; k=20

Query	GT	AA_{20d}	UA_{20}	AUA_{20}	R_{20wd}	P_{20wd}	P_{20d}
1	4	12	9	4	4/4≡100%	4/9≡44.44%	12/20≡60%
2	6	15	9	6	6/6≡100%	6/9≡66.67%	15/19≡78.94%
3	4	12	6	4	4/4≡100%	4/6≡66.67%	12/18≡66.67%
4	6	20	6	6	6/6≡100%	6/6≡100%	20/20≡100%
5	5	20	5	5	5/5≡100%	5/5≡100%	20/20≡100%
6	4	20	4	4	4/4≡100%	4/4≡100%	20/20≡100%
7	6	16	7	5	5/6≡83.33%	5/7≡71.42%	16/20≡80%
8	24	19	14	13	13/24≡54.16%	13/14≡92.85%	19/20≡95%
9	9	20	9	9	9/9≡100%	9/9≡100%	20/20≡100%
10	4	20	4	4	4/4≡100%	4/4≡100%	20/20≡100%

Table 5.7. Experiment findings for top 25 answer nodes; k=25

Query	GT	AA_{25d}	UA_{25}	AUA_{25}	R_{25wd}	P_{25wd}	P_{25d}
1	4	14	12	4	4/4≡100%	4/12≡33.33%	14/25≡56%
2	6	15	9	6	6/6≡100%	6/9≡66.67%	15/19≡78.94%
3	4	12	6	4	4/4≡100%	4/6≡66.67%	12/18≡66.67%
4	6	25	6	6	6/6≡100%	6/6≡100%	25/25≡100%
5	5	25	5	5	5/5≡100%	5/5≡100%	25/25≡100%
6	4	25	4	4	4/4≡100%	4/4≡100%	25/25≡100%
7	6	19	9	5	5/6≡83.33%	5/9≡55.55%	19/25≡76%
8	24	24	14	13	13/24≡54.16%	13/14≡92.85%	24/25≡96%
9	9	25	9	9	9/9≡100%	9/9≡100%	25/25≡100%
10	4	25	4	4	4/4≡100%	4/4≡100%	25/25≡100%

Table 5.8. Recall for Query 11 and Query 12

Query	GT	AA_d	R_d
11	7002	6246	6246/7002≡89.20%
12	7002	6246	6246/7002≡89.20%

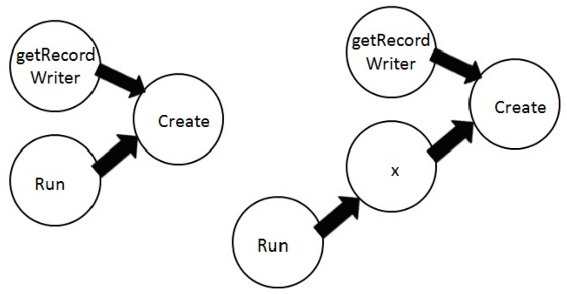

Figure 5.6. Query intent (left) and one inaccurate result (right) for Query: getRecord-Writer run calls create.

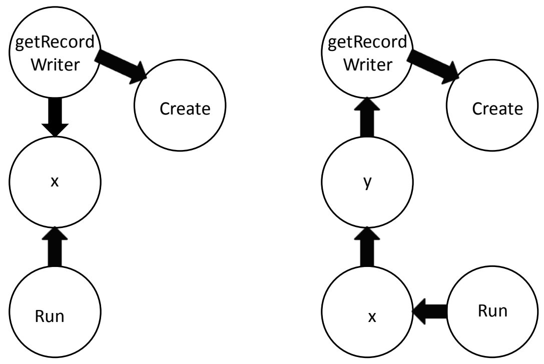

Figure 5.7. Two inaccurate answer patterns for Query: getRecordWriter run calls create.

where "getRecordWriter" calls "create" method, but "run" calls "getRecordWriter" through another method. This has been shown in fig 5.6. Fig 5.7 shows two other inaccurate result patterns that we have noticed for the same query.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

It may be useful if a code search engine may allow free form query to search for structural relationship from the source code. According to our study, the existing code search systems still have not considered that. So we have proposed a code search model which considers structural relationship's name present in the search query to find relevant answers. In addition, we have implemented a prototype of the system. To find out how our system performs, we have come up with some example queries and passed those queries to the system. We have scrutinized each of the answers returned by the system to find out the relevancy with the query intent. We have also measured the precision and recall for our system. Our experiment results shows that our system is fairly successful in finding structural relationships that are relevant to user query.

6.2 Future work

There are a number of areas where we can improve our system. First, our current model does not know how the keywords in the neighborhood of an answer node are actually connected to each other. Our future improvement would be to find a way to solve this problem. Second, the current prototype shows the code snippet for the returned code entity. This may not be ideal. Different queries may require different approaches to show the code snippet. For example, a query "foo calls bar" may show a code snippet where method "foo" calls method "bar". This approach may not work when the query is "foo calls method calls bar". It requires some mechanism to show code snippets corresponding to different types of user queries. This can be a future improvement of our work. Third, our prototype only implements "calls" relationship. Our model proposes some other relationships other than "calls". Our prototype may be extended to support those relationships and see how the model performs. Fourth, we have implemented a code downloader module that only downloads code projects from apache software repository. There are many other software repositories which we do not support yet. These repositories may require a different implementation to successfully download code projects. So another possible expansion of our project could be supporting more software repositories. Last but not the least, natural language processing (NLP) may be useful in interpreting user's intent from the free form query. We may consider integration of NLP to our search module to better understand user's query intent.

APPENDIX A

A WORKED OUT EXAMPLE OF THE SEARCH ALGORITHM

In this appendix, we work out an example to show how our code search algorithm works. We take a small java program. We draw the data graph that corresponds to the program. We show the neighborhood files, index files after applying the algorithm. We take a couple of example queries and show the result node/nodes produced by our algorithm.

A.1 Code snippet

For simplicity, we use the same code snippet we have used in the earlier chapters.

```
package edu.uta.cse.td;
/*@author csallner@uta.edu (Christoph Csallner)*/
public class TipCalc
{
  /*Calculate tip.*/
  public static void main(final String[] args)
  {
    if (args==null || args.length != 2)
    {
     System.out.println("need 2 parameters");
     return;
    }
    final int a = Integer.parseInt(args[0]);
    final int b = Integer.parseInt(args[1]);
    final int res = (a * b) / 100;
    System.out.println(res);
  }
```

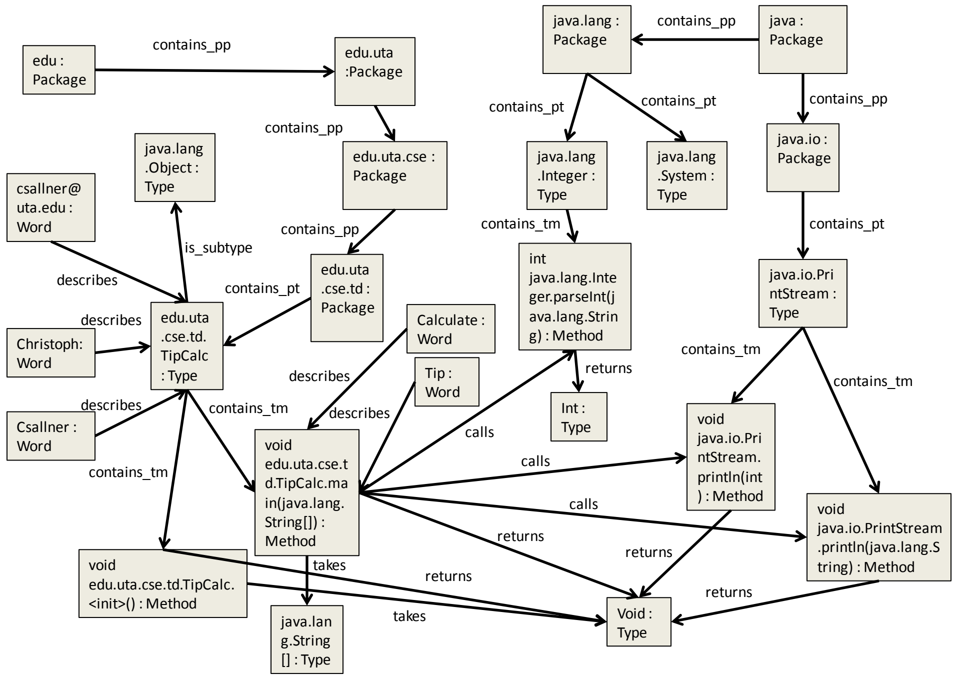

Figure A.1. Data Graph.

}

A.1.1 Data graph

For better understanding of the algorithm, we show the data graph corresponding to the code snippet in figure A.1. The details of data graph have already been discussed in chapter 2, so we leave that discussion out of this appendix.

A.1.2 Neighborhood files

We apply the neighborhood file generation algorithm on the data graph. The algorithm produces a number of neighborhood files. These neighborhood files are listed in Table A.1. Each column header is a neighborhood file name and the content of the file is below the header.

Table A.1. Neighborhood files with content

edu.hood	java.lang.object.hood	edu.uta.hood
(edu , 1) (package , 1) (uta , 0) (contains_pp , 0)	(object , 1) (type , 1) (TipCalc , 0) (Is_subtype , 0)	(uta , 1) (package , 1) (edu , 0) (contains_pp , 0) (cse , 0)
csallner@uta.edu.hood	Christoph.hood	csallner.hood
(csallner@uta.edu , 1) (word , 1) (tipcalc , 0) (type , 0) (describes , 0)	(christoph , 1) (word , 1) (tipcalc , 0) (type , 0) (describes , 0)	(csallner , 1) (word , 1) (tipcalc , 0) (type , 0) (describes , 0)

A.1.3 Index files

We apply the index generation algorithm on the previous neighborhood files. The algorithm generates a number of index files. Instead of showing all the index files, we show only the ones that will be used in this example. Table A.2 and Table A.3 lists those index files. Each column header is an index file name and the content of the file is below the header.

A.1.4 Search

User query1: main calls parseInt

Load idx files: main.idx , calls.idx, parseInt.idx

Result nodes: edu.uta.cse.td.tipcalc.main()

Only one result node hence this is the acceptable node without any ranking.

User query2: package contains tipcalc

load idx files: package.idx, contains_xx.idx, TipCalc.idx

result nodes: edu.uta.cse.td.TipCalc (matches both in contains_pt and contains_tm), edu.uta.cse.td (matches both in contains_tm and contains_pp)

edu.uta.cse.hood	java.lang.string[].hood	calculate.hood
(cse , 1)	(string[] , 1)	(calculate , 1)
(package , 1)	(type , 1)	(word , 1)
(uta , 0)	(main , 0)	(main , 0)
(contains_pp , 0)	(method , 0)	(method , 0)
(td , 0)	(takes , 0)	(describes , 0)
java.hood	java.lang.system.hood	int.hood
(java , 1)	(system , 1)	(int , 1)
(package , 1)	(lang , 0)	(type , 1)
(lang , 0)	(tipcalc , 0)	(parseint , 0)
(contains_pp , 0)	(package , 0)	(method , 0)
(io , 0)	(contains_pt , 0)	(returns , 0)
tip.hood	edu.uta.cse.td.hood	java.lang.hood
(tip , 1)	(td , 1)	(lang , 1)
(word , 1)	(package , 1)	(package , 1)
(main , 0)	(cse , 0)	(integer , 0)
(method , 0)	(contains_pp , 0)	(type , 0)
(describes , 0)	(TipCalc , 0)	(contains_pt , 0)
	(Type , 0)	(java , 0)
	(contains_pt , 0)	(system , 0)
java.lang.integer.hood	java.lang.integer.parseInt().hood	edu.uta.cse.td.TipCalc().hood
(integer , 1)	(parseInt , 1)	(tipcalc() , 1)
(type , 1)	(method , 1)	(method , 1)
(lang , 0)	(Integer , 0)	(tipcalc , 0)
(package , 0)	(type , 0)	(type , 0)
(contains_pt , 0)	(contains_tm , 0)	(contains_tm , 0)
(parseInt , 0)	(int , 0)	(void , 0)
(method , 0)	(returns , 0)	(returns , 0)
(contains_tm , 0)	(takes , 0)	
java.io.hood	java.io.printstream.println.hood	java.io.printstream.hood
(io , 1)	(println , 1)	(printstream , 1)
(package , 1)	(method , 1)	(type , 1)
(java , 0)	(printstream , 0)	(io , 0)
(contains_pp , 0)	(type , 0)	(package , 0)
(printstream , 0)	(contains_tm , 0)	(contains_pt , 0)
(type , 0)	(void , 0)	(println , 0)
(contains_pt , 0)	(returns , 0)	(method , 0)
		(contains_tm , 0)

edu.uta.cse.td.TipCalc.hood	edu.uta.cse.td.tipcalc.main().hood
(TipCalc , 1)	(main , 1)
(Type , 1)	(method , 1)
(td , 0)	(tipcalc , 0)
(package , 0)	(type , 0)
(contains_pt , 0)	(contains_tm , 0)
(object , 0)	(string , 0)
(is_subtype , 0)	(type , 0)
(csallner@uta.edu , 0)	(takes , 0)
(Word , 0)	(void , 0)
(describes , 0)	(returns , 0)
(Christoph , 0)	(println , 0)
(Csallner , 0)	(calls , 0)
(TipCalc() , 0)	(parseInt , 0)
(Method , 0)	(tip , 0)
(contains_tm , 0)	(word , 0)
(main , 0)	(describes , 0)
	(calculate , 0)
java.io.printstream.println(string).hood	void.hood
(println , 1)	(void , 1)
(method , 1)	(type , 1)
(printstream , 0)	(TipCalc , 0)
(type , 0)	(method , 0)
(contains_tm , 0)	(takes , 0)
(void , 0)	(returns , 0)
(type , 0)	(main , 0)
(returns , 0)	(println , 0)

Table A.2. Index files with content

main.idx	contains_pp.idx	contains_pt.idx
edu.uta.cse.td.tipcalc.main() , 1	edu , 0	edu.uta.cse.td , 0
edu.uta.cse.td.TipCalc , 0	edu.uta , 0	edu.uta.cse.td.TipCalc , 0
java.lang.string[] , 0	edu.uta.cse , 0	java.lang , 0
void , 0	edu.uta.cse.td , 0	java.lang.integer , 0
calculate , 0	java , 0	java.lang.system , 0
tip , 0	java.io , 0	java.io , 0
		java.io.printstream , 0

Table A.3. Index files with content continued

package.idx	contains_tm.idx
edu , 1 edu.uta , 1 edu.uta.cse , 1 edu.uta.cse.td , 1 java.lang , 1 java , 1 java.io , 1 edu.uta.cse.td.TipCalc , 0 java.lang.integer , 0 java.lang.system , 0 java.io.printstream , 0	edu.uta.cse.td.TipCalc , 0 edu.uta.cse.td.TipCalc() , 0 edu.uta.cse.td.tipcalc.main() , 0 java.lang.integer , 0 java.lang.integer.parseInt() , 0 java.io.printstream , 0 java.io.printstream.println , 0 java.io.printstream.println(string) , 0
TipCalc.idx	parseInt.idx
edu.uta.cse.td.TipCalc , 1 edu.uta.cse.td , 0 java.lang.object , 0 csallner@uta.edu , 0 Christoph , 0 csallner , 0 edu.uta.cse.td.tipcalc.main() , 0 void , 0 edu.uta.cse.td.TipCalc , 0	java.lang.integer.parseInt() , 1 edu.uta.cse.td.tipcalc.main() , 0 java.lang.integer , 0 int , 0
string.idx	calls.idx
edu.uta.cse.td.tipcalc.main() , 0	edu.uta.cse.td.tipcalc.main() , 0

score(edu.uta.cse.td.TipCalc) = 0 + 0 + 1 = 1 (using contains_pt);

score(edu.uta.cse.td.TipCalc) = 0 + 0 + 1 = 1 (using contains_tm);

score(edu.uta.cse.td) = 1 + 0 + 0 = 1 (using contains_tm);

score(edu.uta.cse.td) = 1 + 0 + 0 = 1 (using contains_pp);

both edu.uta.cse.td.TipCalc and edu.uta.cse.td are acceptable result node.

REFERENCES

[1] O. Panchenko, H. Plattner, and A. Zeier, “What do developers search for in source code and why,” in *Search-Driven Development - Users, Infrastructure, Tools and Evaluation (SUITE)*, New York, NY, 2011, pp. 33–36.

[2] S. Sim, C. Clarke, and R. Holt, “Archetypal source code searches: a survey of software developers and maintainers,” in *IWPC. Proceedings., 6th International Workshop on Program Comprehension.*, 1998.

[3] S. Bajracharya *et al.*, “Sourcerer: a search engine for open source code supporting structure-based search,” in *OOPSLA Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, 2006.

[4] E. Hill, L. Pollock, and K. Vijay-Shanker, “Improving source code search with natural language phrasal representations of method signatures,” in *26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Lawrence, KS, Dec. 2011, p. 524.

[5] S. Bajracharya, J. Ossher, and C. Lopes, “Sourcerer: An internet-scale software repository,” in *Search-Driven Development-Users, Infrastructure, Tools and Evaluation (SUITE)*, Vancouver, BC, Canada, May 2009.

[6] R. Vallee-Rai, P. Co, E. Gagnon, L. H. Patrick, and L. V. Sundaresan, “Soot - a java bytecode optimization framework,” in *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research(CASCON)*, 1999, p. 13.

[7] E. Bruneton, R. Lenglet, and T. Coupaye, “Asm: A code manipulation tool to implement adaptable systems.” in *Technical report, France Telecom R&D*, 2002.

[8] V. Hristidis and Y. Papakonstantinou, “Discover: keyword search in relational databases,” in *Proceedings of the 28th international conference on Very Large Data Bases(VLDB)*, 2002, pp. 670–671.

[9] F. Liu, C. Yu, W. Meng, and A. Chowdhury, “Effective keyword search in relational databases,” in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, 2006, pp. 563–574.

[10] E. Chu, A. Baid, X. Chai, A. Doan, and J. Naughton, “Combining keyword search and forms for ad hoc querying of databases.” in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, 2009, pp. 349–360.

[11] J. Pound, I. F. Ilyas, and G. Weddell, “Expressive and flexible access to web-extracted data: A keyword-based structured query language,” in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010, pp. 423–434.

[12] M. Verbaere, E. Hajiyev, and O. D. Moor, “Improve software quality with semmlecode: an eclipse plugin for semantic code search,” in *Proceeding OOPSLA Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, 2007.

[13] K. D. Volder, “Jquery: A generic code browser with a declarative configuration language,” *Lecture Notes in Computer Science*, vol. 3819/2006, pp. 88–102, 2006.

[14] H. Ghanbari, “A hybrid query engine for the structural analysis of java and aspectj programs,” in *WCRE. 15th Working Conference on Reverse Engineering, 2008.*, Antwerp, Oct. 2008, pp. 133–137.

[15] S. Reiss, “Semantics-based code search,” in *ICSE IEEE 31st International Conference on Software Engineering, 2009.*, Vancouver, BC, May 2009, p. 243.

[16] R. Holmes and G. C. Murphy, “Using structural context to recommend source code examples,” in *ICSE Proceedings of the 27th international conference on Software engineering*, 2005.

[17] I. Keivanloo *et al.*, “Se-codesearch: A scalable semantic web-based source code search infrastructure,” in *IEEE International Conference on Software Maintenance (ICSM)*, Montreal, QC, Canada, Sept. 2010, pp. 1–5.

[18] C. McMillan *et al.*, “Portfolio: finding relevant functions and their usage,” in *ICSE Proceedings of the 33rd International Conference on Software Engineering*, 2011.

[19] O. A. L. Lemos *et al.*, “Codegenie: using test-cases to search and reuse source code,” in *ASE Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, 2007.

[20] S. Thummalapenta and T. Xie, “Parseweb: a programmer assistant for reusing open source code on the web,” in *ASE Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, 2007.

[21] S. Chatterjee, S. Juvekar, and K. Sen, “Sniff: A search engine for java using free-form queries,” *Lecture Notes in Computer Science*, vol. 5503/2009, pp. 385–400, 2009.

[22] J. Kim, S. Lee, S. won Hwang, and S. Kim, “Towards an intelligent code search engine,” in *Twenty-Fourth AAAI Conference on Artificial Intelligence*, Atlanta, Georgia, USA, July 2010.

BIOGRAPHICAL STATEMENT

Asheq Hamid was born in Dhaka, the capital of Bangladesh, a small beautiful country in South East Asia. He completed his Bachelors degree from Bangladesh University of Engineering and Technology (BUET) in Computer Science and Engineering. After completing his Bachelors degree, he contributed to an open source localization project for one and half year. He started his Masters in University of Texas at Arlington in Fall 2010. He joined Software Engineering Research Center (SERC) Lab in Spring 2011. His research and professional interests are mainly on software engineering.