Dependency Extraction
-
MySQL Dependencies

EECS 4314 - Advanced Software Engineering
Fall 2017
York University

Team

1. Benjamin Zaganjori (212987533)
2. Dan Sheng (213587712)
3. Daiwei Zhou (209112905)
4. Hassaan Ahmed (213155700)
5. Matthew MacEachern (213960216)
6. Shagun Kazan (213741707)
7. Tushar Chand (213258009)

Table of Contents

Abstract

This report provides insight into dependency extraction, with a focus on three different tools: Scitools Understand, SrcML, and a custom dependency extraction tool implemented by the group members. Additionally, the accuracy of each tool was compared with one another. The MySQL source code root folder was provided to each tool in an attempt to parse and retrieve dependencies between files contained within all the subdirectories. Once complete, the results presented by the three tools were compared using mathematical precision and recall.

Once this process was completed it became apparent that Understand captured the most dependencies, while the custom extraction method caught the least, and rightly so. Understand is a professional, black boxed, licensed, enterprise software dependency extraction tool while the Java program only looks for dependencies within C, C++ and Java extension files. Interestingly, the group's program missed about 2300 dependencies when compared to the open source implementation SrcML, which is not a very large number for a system of this size. Understand found slightly less than double SrcML's extraction results.

These results show that the group's implementation is not far off from a trusted open source dependency extraction tool, however the professionally licensed software does a far better job at locating and mapping dependencies between files through deeper searches within function calls and variable usage.

1. Introduction

Dependency extraction is an invaluable tool when trying to understand a large scale (and even small scale) software system. Dependency extraction as a process not only allows developers and analysts to gain further knowledge into systems that they may not have had a great understanding of previously, but also provides these individuals with yet another metric to evaluate their software system. A dependency occurs when one file in a software system uses another file in a software system (whether it is internal to the software system it is contained in, or external to the software system it is contained in) in some way. These dependencies give developers a general idea about which systems and subsystems are using other subsystems throughout the software project and can give a good idea of how important some subsystems are.

Although not discussed in length in this paper, dependency extraction can provide a wealth of statistical information about the software system given the write data analysis tools. For example, if a developer wanted to find out how many files depend on (ie. have a dependency to) a certain file, they could first run the dependency extraction tool of their choice, and then query the data produced by that file to see how many dependencies points to that file. This is just one of the many examples of the insight that can be extracted from the dependency list of a software system.

There are many different methods when looking at completing the task of dependency extraction. These different methods are not always guaranteed to give the same results as each other, and very often they do not give the same results. As will be discussed throughout this paper, Understand, SrcML, and Include directives are all tools and techniques that can be used for dependency extraction and with them bring their own unique benefits and shortcomings.

2. Dependency Extraction using Understand

2.1 Overview of SciTools Understand

SciTools Understand is a powerful tool developed by Scientific ToolWorks at St. George, Utah. It is a customizable Integrated Development Environment (IDE) with static code analysis features while also helping developers maintain, document and understands millions of lines of complex source code written in various programming languages. It accomplishes this by deriving and exporting visuals, different types of documentation and by employing the metric tools within the suite. Understand is a complex, licensed piece of software used by developers and industries across the world serving commercial and academic purposes. The tool has several highlighted features such as code knowledge, dependency analysis, metrics and reporting, code editor, graphing, search, standards testing and various language support. The main features of Understand used in dependency comparison are dependency analysis, metrics and reporting. [1]

Reporting:

Cross-reference **Structure report**

Quality report **Metrics report**

Code Metrics

Complexity **Volume**

Object Oriented

One method of dependency analysis within Scitools Understand are dependency graphs. Using the source code, Understand can produce a graph that maps the dependencies from one module to another. Unfortunately, this graph can be incredibly chaotic for larger systems - which MySQL is. Some tools are provided to clarify the graph further. For example, one could hide several selected nodes or highlight all outward edges if looking for particular details. The dependency graph was not very useful for dependency extraction of MySQL, however it is possible that it could be very helpful to view dependencies in other, smaller systems.

2.2 Extracting Dependencies with Understand

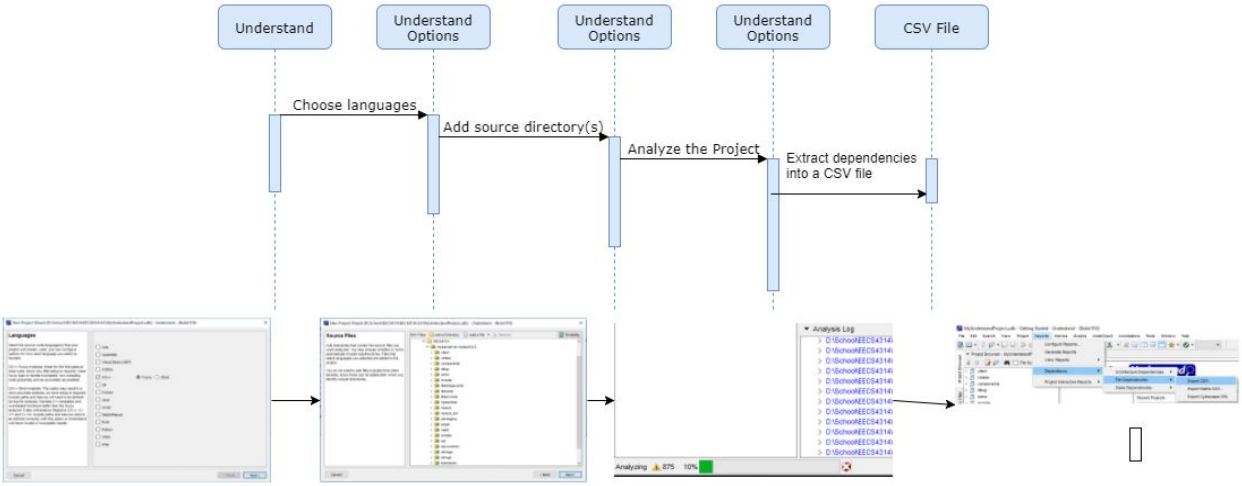
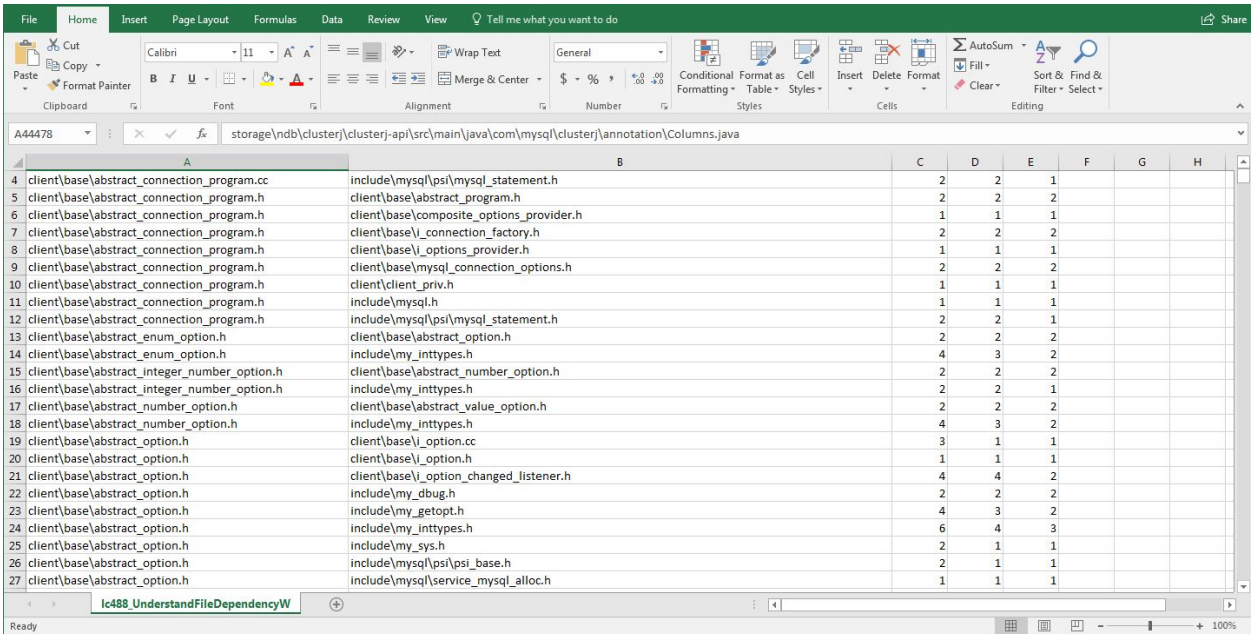


Figure 1. Sequence diagram showing how the Understand extraction program operated.

The use of the Understand tool for dependency extraction was done by first going through the menu and selecting File->New->Project and inputting the name of the project and path that the understand project will be created in. From there, the language selection was completed, which consisted of C/C++ as well as Java. These languages were selected due to insight provided from the previous assignment; C, C++, and Java files were found in the source code folder. The next step was to select the actual source code folder of MySQL in order for the Understand tool to begin its analysis. When the analysis was complete, the results tabulated in the bottom of the window showed that Understand had analyzed a total of 6329 files.

Figure 2. Understand analysis log to show the total number of files analyzed.

In order to acquire the dependencies of these files, the selection from the menu of Reports->Dependency->File Dependencies->Export CSV was performed and the "Relative Name" option was selected from the window that was created. This created a .csv file that was opened with Excel and allowed the group to determine the dependencies that Understand had extracted from the provided MySQL source code folder. The determination of the total number of dependencies was from checking the final row number in this spreadsheet and subtracting 1 from it due to the top row consisting of column headers. From this, it was determined that there were 72983 dependencies from the Understand tool.



	A	B	C	D	E
4	client\base\abstract_connection_program.cc	include\mysql\psi\mysql_statement.h	2	2	1
5	client\base\abstract_connection_program.h	client\base\abstract_program.h	2	2	2
6	client\base\abstract_connection_program.h	client\base\composite_options_provider.h	1	1	1
7	client\base\abstract_connection_program.h	client\base\i_connection_factory.h	2	2	2
8	client\base\abstract_connection_program.h	client\base\i_options_provider.h	1	1	1
9	client\base\abstract_connection_program.h	client\base\mysql_connection_options.h	2	2	2
10	client\base\abstract_connection_program.h	client\client_priv.h	1	1	1
11	client\base\abstract_connection_program.h	include\mysql.h	1	1	1
12	client\base\abstract_connection_program.h	include\mysql\psi\mysql_statement.h	2	2	1
13	client\base\abstract_enum_option.h	client\base\abstract_option.h	2	2	2
14	client\base\abstract_enum_option.h	include\my_inttypes.h	4	3	2
15	client\base\abstract_integer_number_option.h	client\base\abstract_number_option.h	2	2	2
16	client\base\abstract_integer_number_option.h	include\my_inttypes.h	2	2	1
17	client\base\abstract_number_option.h	client\base\abstract_value_option.h	2	2	2
18	client\base\abstract_number_option.h	include\my_inttypes.h	4	3	2
19	client\base\abstract_option.h	client\base\i_option.cc	3	1	1
20	client\base\abstract_option.h	client\base\i_option.h	1	1	1
21	client\base\abstract_option.h	client\base\i_option_changed_listener.h	4	4	2
22	client\base\abstract_option.h	include\my_dbug.h	2	2	2
23	client\base\abstract_option.h	include\my_getopt.h	4	3	2
24	client\base\abstract_option.h	include\my_inttypes.h	6	4	3
25	client\base\abstract_option.h	include\my_sys.h	2	1	1
26	client\base\abstract_option.h	include\mysql\psi\psi_base.h	2	1	1
27	client\base\abstract_option.h	include\mysql\service_mysql_alloc.h	1	1	1

Figure 3. The .csv spreadsheet originally created and displayed in Excel

The conversion of this data into a raw text file was performed by first creating a new column between A and B in this spreadsheet with the string of '->', and then combining the columns of A, '->', C into a single column in the excel sheet. Then all the rows that were not the first row were copy and pasted onto a new text file.

The data in column C was modified in order to ensure that only the filename was displayed instead of the entire path of the file within the MySQL software system. This was a design decision that was chosen due to the simplicity of only having to check the filename instead of the path due to the way our two other dependency extraction techniques handled filenames (see section 3 and 4 for further details).

3. Dependency Extraction by Include Directives

3.1 Introduction

One of the dependency extraction techniques used in this assignment was to use the include directives in the files included in the MySQL project structure. In order to do this, every file in the entire MySQL directory would need to be scanned and parsed in order to extract which other files (internal to MySQL or external) the project file depended on. In order to accomplish this, whenever a '#include' statement was found in the case of .c, .cpp, .cc, .hpp, and .h files, the file name after the include statement would be part of the list of dependencies associated with the current file that was being scanned. It was discovered in assignment 2 that the MySQL project did not only contain C and C++ files, but also included Java files. This fact was kept in mind when writing the program to discover the dependencies via include directives.

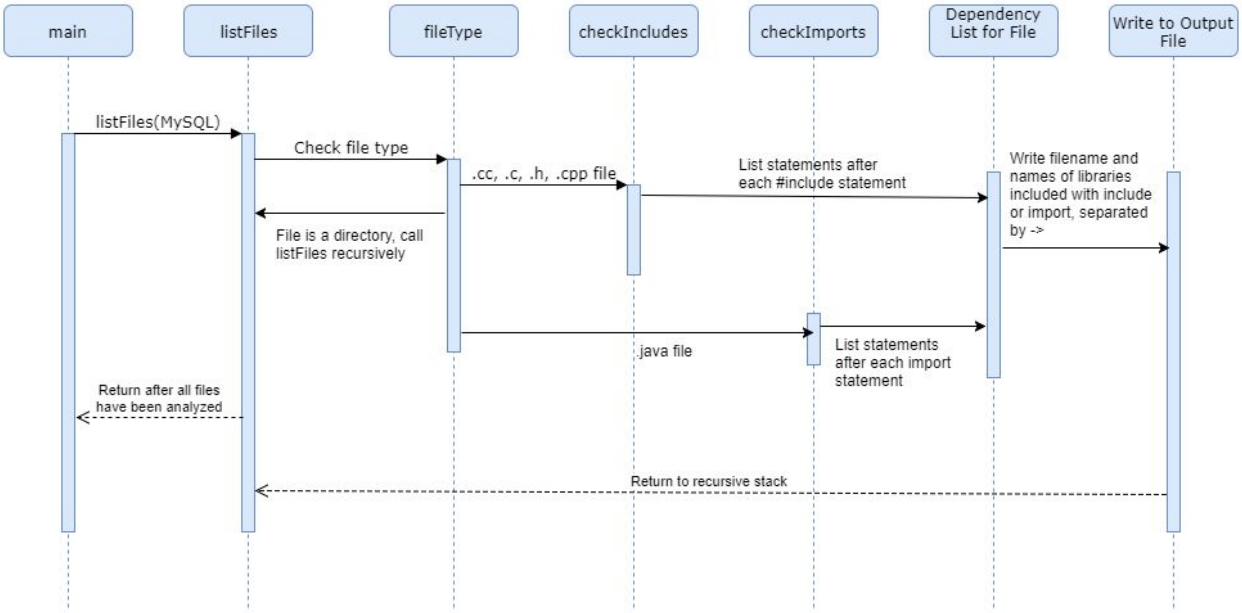


Figure 4. Sequence diagram showing how the include directive extraction program operated.

3.2 Program Operation

The program written to accomplish this dependency extraction task was written in Java, and included a simple recursive algorithm to scan through the entire directory structure of MySQL. As shown in Figure 4, the entire MySQL directory structure was first fed into the program in order to begin the extraction process. Then, the program would do a simple check: if the file encountered was a directory, another recursive call was made in order to go into that directory, ensuring that all files in the directory structure were scanned properly. However, if the file was a .cc, .c, .h or .cpp file, the file was opened and scanned line by line. If a line was found that began with the '#include' statement, the line was parsed so that only the filename of the library or file was written to the output file (our TA file). The current filename and the filename that the current file included was written to the output file and the next statement was parsed. Similarly, as mentioned above the MySQL project included many Java files. If a Java file was encountered, the same process would be executed, except with the 'import' statement being checked for in lieu of the '#include' statement. Due to the recursive nature of the program, when a directory had been entirely scanned, the program would return to its recursive stack in order to further scan the rest of the directories contained within it. This ensured that the entire directory structure was parsed, and no files were left unscanned unless the program didn't explicitly name it to be parsed.

3.3 Benefits and Shortcomings of this Method

Extracting the dependencies of a software system using the include directives contained within the source files of the system is a good way to get a very general overview of what systems depend on others in the software system. It is not without its shortcomings, however, and it should not be used if a higher granularity is required for the software system.

One huge benefit of the include directive method of extracting dependencies from a system was speed. With the program that was created for this assignment, it managed to run through and parse the MySQL software system (a system with thousands of files and thousands of lines per file to scan through) in under a second. Another benefit of this method lies in its simplicity. There is no third party software required in order to understand how this method works. The program simply takes the files that are listed in the include statements, and links them in a way to the file that they are contained in. This simplicity also allows for the modification of the program such as to add or remove filters for file types in order to parse through additional files as seen fit.

There are inherent downfalls that come with the simplicity of this method. Only showing dependencies of a system based purely on the include statements within the source files of the system does not show the full picture of the software system itself. For example, the include statements only show dependencies from the .cc, .c, .cpp, and .h files to other .h files or external libraries such as string, vector, etc. This is because in a C or C++ source file, include is used to import the header files of various files into the system. This means that certain dependencies that occur because of inheritance and other means are not shown in the output of this method and are essentially lost.

Another shortcoming of this method is that include statements don't necessarily exhibit the true dependencies within the system based on the header statements. For example, a developer includes a library in source file A that links to another source file B, uses the functions from B for some time but decides to refactor A at a later date such that A no longer uses the functions from B. If the developer is no longer using the functions due to the refactor but forgets to redact the include statement, the dependency list will show outdated and therefore inaccurate dependencies.

3.4 Quantitative Data for the Include Method

With the include directive method of dependency extraction, a total of 5688 files were scanned. This included .h, .c, .cpp and .cc files. Out of those 5688 files, a total of 42127 dependencies were extracted, meaning that a total of 42127 include or import statements were encountered during the parsing of the files due to the nature of the program that was written for this method.

4. Dependency Extraction through srcML

4.1 Overview

Source Markup Language (srcML) is a tool developed in part by Michael Collard and Jonathan Maletic at Kent State University, and currently only supports C/C++, C#, and Java. Its main purpose is to process source code into XML format with Abstract Syntax Tree tags (Class > Method > Variable) and attributes. All files within the source code are tagged, have identifying attributes (file path, language, hash, etc.) and nested within are the tags for syntax keywords and operations (imports, defined functions, variables, etc.). Organizing source code in an XML format allows for easy exploration, analysis, and manipulation as seen in our usage. [2]

While exploration and analysis are supported by exploring the XML file itself, manipulation is done through another feature. Reversed operation of srcML is supported, wherein source code that has been organized into the XML format by srcML can be rebuilt into the original structure (folders and files).[2] However, such a feature is out of the scope for this assignment and is not used.

In conclusion, srcML is a powerful tool for organizing source code into an easily parsable format for purposes of exploration, which is why the tool was chosen as the third method for dependency extraction.

4.2 Dependency Extraction by srcML

With the tool installed, the first step was to run the program passing in the source code folder, an output flag, and the output file. The command executed is shown in the figure below:

```
C:\Users\dave\Desktop\EECS 4314\mysql-server-mysql-8.0.2>srcml mysql-server-mysq
l-8.0.2 -o mysql.xml
```

Figure 5. srcML command executed in command prompt to generate XML file of source code

The execution of this command created a file called mysql.xml that was generated from the input of the MySQL source code folder. The file itself was very large (~487 MB, ~3.7 million lines) as it contained all the contents from the MySQL source code, as well as tagging information. To retrieve a usable list of dependencies from the XML file, a bash script was written (xml_parse.sh) The script can be found here.

Investigating the file revealed that the output XML file had the <unit> tag for all source files (Java and C). Within each <unit> tag held attributes of the source file such as the hash, language, and path of the source file. The notable attributes were the language and filename attributes. The language attribute was used to differentiate between the Java and C files, while the filename allowed differentiating between the type of source file, as well as building the output into the required .ta format. For the C/C++ files, all file extensions (.c, .cpp, .cc, .h, .hpp) were categorized under either “C” or “C++”, so a simple regex was used to catch them all. Search and extraction was done using regex and cut.

Contents of the file were kept between the <unit> and </unit> tags, with the key tags considered in this extraction being <cpp:include> for C/C++ files, and <import> for Java files. Within the <cpp:include> tag, the dependency file was held between a <cpp:file> tag. All that was required was a simple extraction and clean up. Java dependencies were similar as the line was contained within <name> and <operation> tags, which were simply removed to retrieve the necessary Java dependencies.

For every new dependency found, the dependency and associated source file name (found from the filename attribute in containing tag <unit>) were written to an output file of the .ta format (File -> Dependency).

4.3 Reflection on srcML

While srcML is a powerful tool for exploring and analyzing source code, it was not as useful for extracting dependencies between source files. Function calls and variable usage did not show the deeper connections between source files and their associated dependencies. The process of parsing the generated XML was more akin to finding dependencies through the "#include" directives. This is because srcML does not process the source code and link files to each other, but only adds explicit tags to syntax.

Furthermore, there were difficulties working with the generated XML as the file created was significantly large (~487 MB, ~3.7 million lines), and resulted in issues when opening and scrolling through the file. Parsing with the script was also time consuming (~15 minutes). This is likely due to the addition of XML tags and attributes which consisted of over half the document.

5. Quantitative Comparison

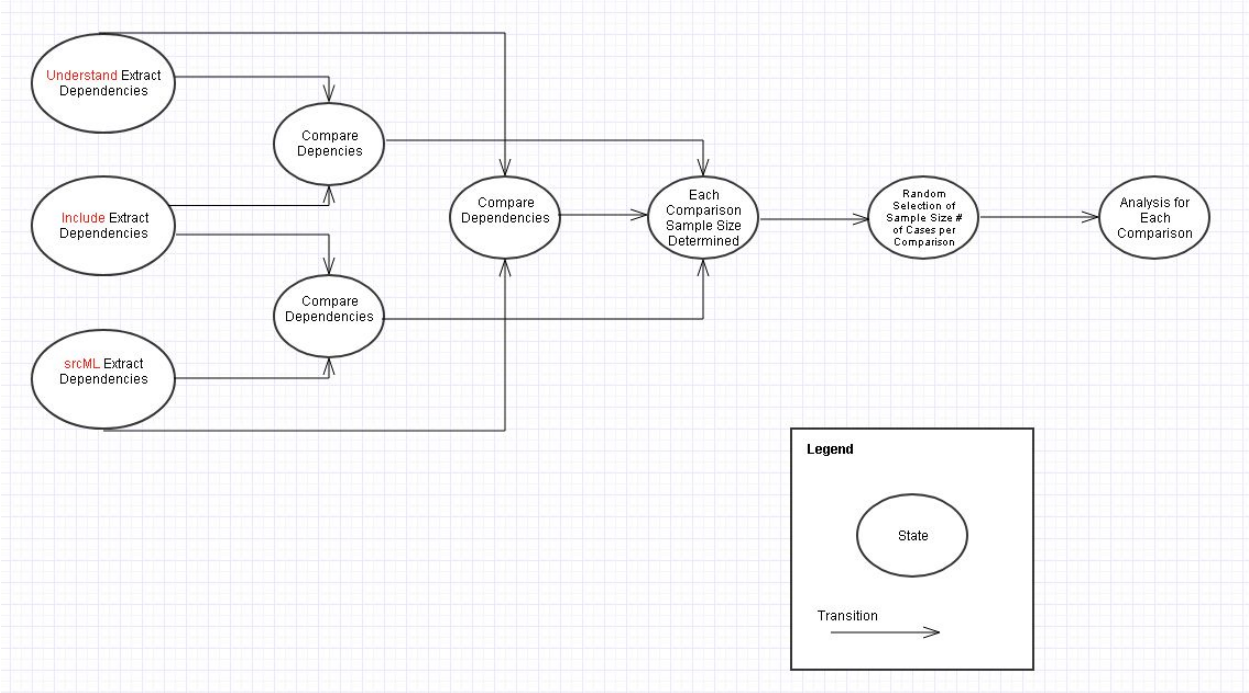


Figure 6. State diagram detailing the process for each comparison of the extracted dependencies..

5.1 Technique Used to Compare Dependencies Between Methods

In order to compare the differences between the dependencies extracted using the various techniques presented thus far, a program had to be created that would compare line by line each dependency. In order to accomplish this, compareDependencies.java was written. This program takes an input of two files (in this case the two files were the two dependency output files that needed to be compared). The program would then count the total number of dependencies in both files. In order to facilitate this, the dependency files that were used as input to this program were formatted in such a way that one dependency was on each line. Therefore the number of lines in the file was the number of dependencies. The file with more dependencies was then used as the starting file to compare the other file to. This would ensure that each line in the larger file was parsed through and the number of unique dependencies in the second file could be inferred by knowing how many matching dependencies are between the first file and the second.

This method of comparison runs in O(n^2) time, which allows the program to run in approximately five minutes depending on which dependency files are being compared. Figure 7, Figure 8, and Figure 9 show the number of common dependencies between the three techniques as well as the number of unique dependencies between them.

5.1.1 Potential Risks and Limitations Associated with the Technique Used

The technique described in section 5.1 is sufficient for our use but it does have limitations and potential risks associated with it due to certain assumptions being made in the writing of the comparison program (compareDependencies.java). For instance, the program assumes that the dependency files are both laid out in the exact same manner, since the program is doing a simple string comparison between lines of the two files. This can be restrictive due to how the dependencies are extracted in the first place. If one method extracts the entire file path rather than the file name, the dependency will not be matched and it will show as a unique dependency in the comparison process.

Another limitation associated with this technique comes from the way file paths are specified on different operating systems. For example, macOS and Linux specify file paths using the forward slash while Windows uses backwards slash to specify file paths. This was a minor annoyance when dealing with the different extraction methods due to the fact that different team members developing the programs to parse the dependencies were running different operating systems and the output of their dependency files were different when comparing them to others. This was a fairly simple fix, however a limitation nonetheless.

Finally and perhaps most significant of all limitations is the run time of the comparison process, which is O(n^2). This was significant when running the program, especially with how large the Understand dependency file was (72,983 lines). The program would take a significant amount of time to parse the data in each files, with a worst case run time at roughly 7 minutes running on a computer with a SSD (meaning faster reads). Although not tested, it would most definitely take much longer on commodity hardware and computers running standard HDDs.

5.2 Understand Technique vs. Include Directive Technique

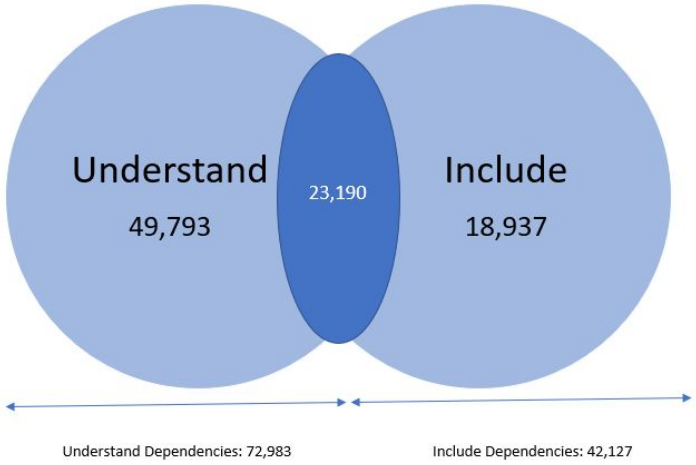


Figure 7. Comparison of Dependencies Extracted Using Understand and Include

Using the technique described in section 5.1, the number of common dependencies found between the include directive method and the Understand method was 23,190. This meant that roughly 55% of the dependencies extracted using the include directive method were shared with the dependencies extracted using the Understand method. This provided a total of 91,920 dependencies in total between the two techniques. Refer to Figure 7 for a visual description of the data.

5.3 Understand Technique vs. SrcML Technique

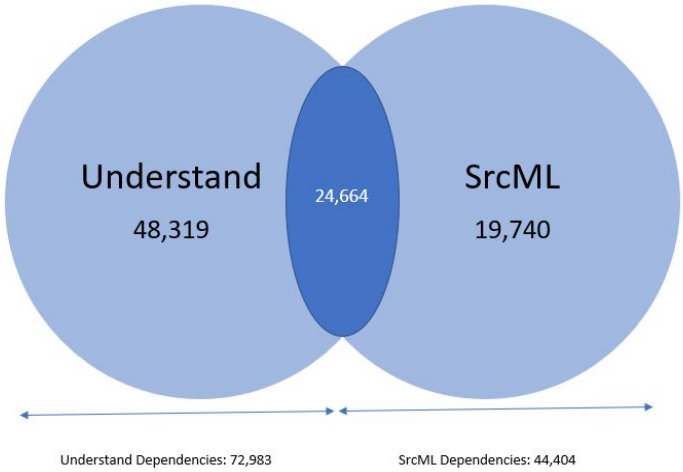


Figure 8. Comparison of Dependencies Extracted Using Understand and SrcML

Using the technique described in section 5.1, the number of common dependencies found between the SrcML method and the Understand method was 24,664. This meant that roughly 55% of the dependencies extracted using the SrcML method were shared with the dependencies

extracted using the Understand method. This gave provided a total of 92,723 dependencies in total between the two techniques. Refer to Figure 8 for a visual description of the data.

5.4 Include Directive Technique vs. SrcML Technique

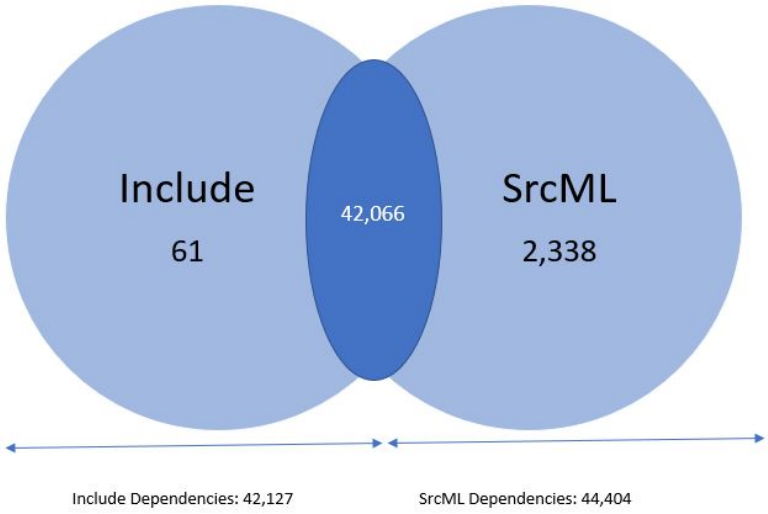


Figure 9. Comparison of Dependencies Extracted Using Include and SrcML

Using the technique described in section 5.1, the number of common dependencies found between the SrcML method and the include directive method was 42,066. This came as no surprise since the method of extracting dependencies using both of these techniques was very similar. This meant that roughly 95% of the dependencies extracted using the SrcML method were shared with the dependencies extracted using the include directive method. This gave a total of 44,465 dependencies in total between the two techniques. The discrepancy between the number of dependencies is explained in section 3.3 as the include method (and by association the SrcML method), does not give the full picture of the dependencies in the software system. Refer to Figure 9 for a visual representation of the data.

5.5 Implications of the Comparison and Recommendations

It can be observed from the gathered information that the Understand tool had significantly more dependencies that were extracted due to the nature in which it gathers them. The methods used for Include and srcML didn't account for the underlying dependencies such as the .cc to .cc file relationships, or the non-direct dependencies that occur due to a file that is depended on from one dependency aren't directly quoted as an "include" in another file. The srcML extraction method could have been performed better to also gather the <function> calls in the xml file to help trace these indirect dependencies. The inclusion of the Java files also brings into question if the idea to also include the python files in the source code folder to the group's

consideration of extraction and comparison, but the decision was made not to due to the small amount of them as well as the added complexity.

6. Qualitative Comparison of Extraction Methods

In order to perform a qualitative analysis on the data extracted with the three methods mentioned previously, the dependency files were imported into an Excel spreadsheet in order to sample from each of the data sets. In order to calculate the sample size that would be required to gain a confidence interval of 95% with a confidence interval of 5%, the calculator available on https://www.surveysystem.com/sscalc.htm was used and it was determined that with the amount of dependencies that were present in each of the three data sets, an appropriate sample size would be 381.[3] The rand() function in Excel was used in order to map a random number to each of the dependencies in each data set. The data sets were then reorganized in their random number assignment order. The first 381 dependencies were chosen to be our sample for each of the data sets. Then, the COUNTIF function was used in order to check if the sample dependency was present in the data set that the sample was being compared to.

6.1 Comparing SrcML and Include

Out of the 381 sampled dependencies from the SrcML data, 370 were found in the include dependency data. Out of the 381 sampled dependencies from the include data, 380 were found in the SrcML data. This gives the following results for precision and recall for SrcML when comparing to include:

$$precision = \frac{370}{370+1} \times 100\% \approx 0.99730458$$
$$recall = \frac{370}{370+10} \times 100\% \approx 0.97368421$$

The following were the values when comparing include to SrcML:

$$precision = \frac{370}{370+10} \times 100\% \approx 0.97368421$$
$$recall = \frac{370}{370+1} \times 100\% \approx 0.99730458$$

7. Conclusion & Lessons Learned

Although dependency extraction and code analysis tools are great, they are still rough around the edges and somewhat incomplete. For instance, Understand takes into consideration various dependencies, such as, include dependencies, variable dependencies, functional dependencies, etc. while srcML yielded similar results to our own implementation. It can be concluded that srcML bases its dependency extraction methods on things such as `import` and `include` statements, similar to what was accomplished in the custom implementation using the include method. Even with such tools as Understand and srcML and features like graphing, report and analysis, understanding vast systems such as MySQL continues to be a colossal task. These features are still unrefined and will get better over time as new versions of these tools continue to be shipped out. For instance, srcML produced a 500mb XML file that describes the MySQL root folder, subdirectories and various files which needs to be parsed by external software such as `xpath` or similar scripts, to gain understanding of the XML files structure. Unfortunately, for the purpose of this experiment the original zipped source code file is a much smaller 80mb contained almost the same amount of information needed. SrcML wraps the original code with hundreds of thousands of XML tags that enables its most important feature - being able to go from source code to XML and back to source code without data loss. [2]

As was explained throughout the report, not all dependency extraction methods will yield the same results, which is a major concern -- what piece of software would a developer or firm choose to help its employees understand a big system are some questions that arise. What is certain is that dependency extraction techniques and tools will only get better over time, even if analysis and understanding of large systems will remain a daunting task.

References

[1] SciTools. “Scitools Understand Features”. Internet: https://scitools.com/features/. Unknown. [Nov. 9, 2017]

[2] srcML. “srcML About”. Internet: http://www.srcml.org/about.html. Unknown. [Nov. 9, 2017]

[3] Creative Research Systems. “Sample Size Calculator” https://www.surveysystem.com/sscalc.htm. Unknown [Nov. 11, 2017]