Improving Speed and Interestingness of Graph-Based Specification Mining via Corpus Filtering

Andrew Mason
ajm188@case.edu

Jon Pfeil
jwp69@case.edu

December 8, 2014

Contents

1 Introduction

Software verification, the ability to prove the correctness of a piece of software [3], is an appealing approach to improving software quality. However, the requirement of having formal descriptions of correctness has prevented the widespread use of software verification [1]. *Specification Mining*, as described in [1], is the machine learning approach to discovering formal specifications for a program interface or an abstract data type. A specification mining tool would reduce the major barrier to program verification by automating the generation of formal specifications.

Research has been done to apply mining techniques to graphical representations of programs, known as Program Dependency Graphs (PDGs). Our project aims to push forward the current state of graphical specification mining through the application of corpus-based filtering. The idea behind this process is that if a pattern appears in almost every project, it is most likely due to boilerplate bytecode, rather than an interesting specification. We will refer to frequent patterns that appear in a percentage of projects higher than a given threshold as "trivial". Frequent patterns that are under the corpus threshold will be referred to as "non-trivial". By ignoring patterns that are trivial, we will increase the interestingness of the patterns we return. By greedily removing disconnected trivial patterns from our graph before applying mining algorithms, we can reduce our input size and thereby reduce the running time of our mining algorithms.

2 Background

2.1 Software Verification

Knowing that a software system fulfills a set of requirements correctly is an important concern in industry, especially for mission-critical software systems, such as the avionics software in an airplane, or the software in medical devices. Approaches to software verification can be separated into categories, dynamic approaches and static approaches.

Dynamic approaches to software verification include unit testing, integration and system testing, and acceptance testing. While these methods can identify bugs in the software and help improve software quality, alone they

cannot confirm that the system is bug free and that the system fulfills all of its requirements. A requirement may not be exercised by the tests, and there may be a bug that occurs only on a particular input which the test suite does not include.

We prefer instead the static methods of software verification, known also as program analysis or formal verification. These techniques aim to provide a formal proof that a system satisfies a specified property. Numerous approaches to formal verification exist, and we briefly highlight a few of them below.

Model-based certification aims to provide formal proofs that an abstract model, such as a formal computational model, representing the software system has a particular property[3]. In this approach, a verifier uses a model-checking tool to explore the state space of the program, searching for error states, states where a certain property of the system does not hold. One issue with this approach is that the state space for even trivial programs can grow quite large, but predicate abstraction can be used to systematically reduce the state space[3].

Deductive software proving, also referred to as program proving, aims to express the correctness of a program as a set of mathematical statements, which are then discharged using a theorem prover [5]. To do this, a formal specification language must be defined and integrated with the desired programming language. Fillitre claims that the best method for achieving this is to tighten a specification language to a programming language within a dedicated program logic, which mixes programs and logical statements, infusing program contstructs with logical ones in a single language [5].

2.2 Specification Mining

Formal software verification has the potential to replace testing as the standard method of verifying program correctness. However, formal program verifiers are limited in that they can only check specified properties. Additionally, authors Lo and Khoo claim that the difficulty in formulating a set of formal properties has contributed to the lack of widespread adoption in industry[7]. To solve this issue, the technique of specification mining aims to discover non-trivial properties or specifications in a software system to later

be used by a program verifier.

Several approaches to specification mining are described in the literature, the first of which is a machine learning approach outlined by Ammons et al. Their mining approach is based on program execution traces rather than static analysis of source code. Ammons et al argue that this approach is superior since the traces will only contain the feasible paths of execution. Additionally, basing the mining on program traces ensure that only correct execution paths are mined, thus preventing buggy paths from polluting the results of the mining[1]. Their miner extracts abstract scenarios from traces annotated with information about flow dependence, which are small sets of independent interactions. These scenarios are then fed into an automaton learner to learn the specifications. Ammons et al give two reasons for this approach. First, the scenarios are much smaller than the original traces, so the automaton learner learns the specifications much faster. Second, the sizes of the scenarios are bounded, which they argue make the specification mining tractable. The workflow of the miner looks like [1]:

Lo and Khoo expand on the work of [1] by describing their work in automaton-based specification mining, as well as their work in mining Linear Temporal Logic (LTL) expressions and Live Sequence Charts (LSC), which are alternate formalisms for representing specifications [7]. Their approach also relies on execution traces of the program. Lo and Khoo improve on [1] by using a data mining approach that attempts to improve the quality of results, since Ammons et al rely on the notion that traces give strong hints about program rules, but do not really address the possibility that these traces may contain errors. To this end, Lo and Khoo add an accuracy metric to measure the performance of an automaton-based miner, which is measured by the notion of recall and precision. They define recall and precision as "the proportion of sentences in S_{inf} that is accepted by S_{orig} and the proportion of sentences in S_{orig} that is accepted by S_{inf} where S_{orig} is the original specification and S_{inf} is the inferred specification [7]." Next, the authors devise a framework they call SMArTIC that splits the specification mining task into four pipelined components. They claim that this improves the quality of the mining because it identifies and filters out erroneous traces early in the mining, and "the over-generalization that occurs at the learning stage can be mitigated by localization of the learning process to groups of related ... traces [7]."

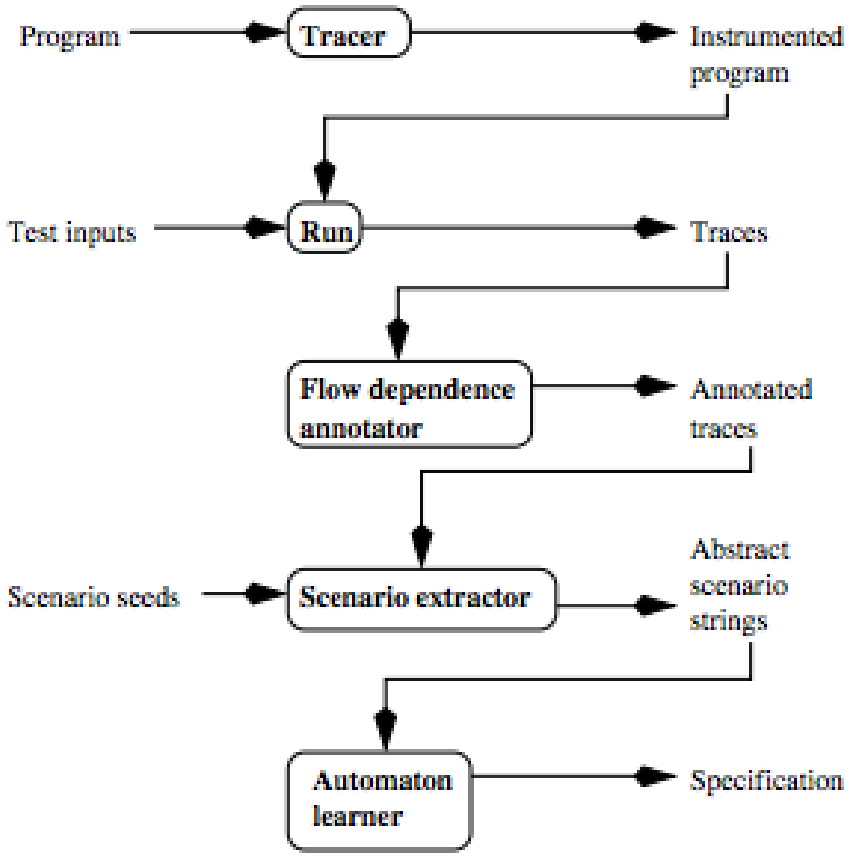

Figure 1: Workflow of machine learning specification miner

Lo and Khoo also discuss their work in LTL specification mining, which is one of the most commonly-accepted formalisms by standard verifiers. LTL expressions break automata-based specifications (which are global and often complex) into smaller pieces which express what the authors call strongly observed behavior or rules. To identify these rules, Lo and Khoo introduce the notions of support, which is the number of traces exhibiting the premise of the rule, and confidence, which is the likelihood of the premise of the rule to be followed by its consequence[7]. They employ a search space pruning strategy to mine these rules from the execution traces.

2.2.1 Limiting False Positives

Le Goues and Weimer[6] claim that specification inference techniques suffer from 90-99% false positive rates, and thus require a heavy burden of manual inspection, making these techniques impractical for most software projects. By defining the notion of code trustworthiness, they are able to achieve false positive rates of only 5%. Their approach statically estimates the trustworthiness of each code fragment and then weights the contribution of each trace by its trustworthiness, which is determined by the trustworthiness of the fragments visited in the trace, in the actual specification mining[6]. The code trustworthiness is based on the notion that some code is more likely to be correct than others. Le Goues and Weimer estimate this likelihood by using the results of many studies in software engineering. For example, studies have shown that code that is modified more frequently is more likely to contain defects, so the approach looks at information obtained from the version control system to determine the rate of what the authors call code churn [6].

2.3 Graphical Specification Mining

2.3.1 Program Dependence Graphs

Several techinques have been described in the literature that perform specification mining on an intermediate representation of the software known as a procedure dependence graph (PDG). A PDG combines the information contained in a data dependence graph and a control flow graph. For more information, see [4].

2.3.2 Subgraph Isomorphism

Finding repeated patterns in the PDGs is central to the ideas behind graphical specification mining, and, therefore, so is the subgraph isomorphism problem. The subgraph isomorphism problem asks the question: given graphs G and H, does there exist a subgraph G_0 of G such that G_0 is isomorphic to H? Such an isomorphism is illustrated in the following diagram (C highlights the subgraph of B that is isomorphic to A):

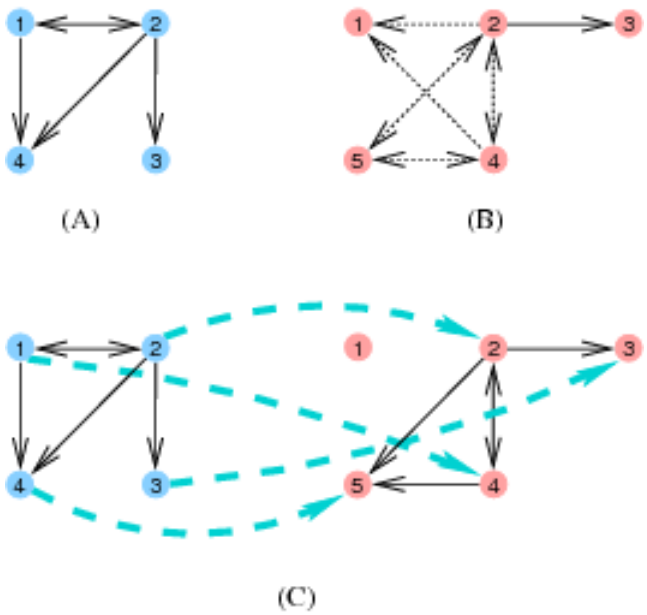

Figure 2: Example of a subgraph isomorphism

2.3.3 Frequent Subgraph Mining

Another fundamental technique in graphical specification mining is that of frequent subgraph mining. Given a graph dataset $D = \{G_0, G_1, \ldots, G_n\}$, with $support(g)$ denoting the number of graphs in D in which g is a subgraph, frequent subgraph mining aims to find all subgraphs g such that $support(g) \geq minSup$, where $minSup$ is some minimum support, usually specified by a user [8]. At the heart of frequent subgraph mining is the subgraph isomorphism test.

2.3.4 gSpan

Yan and Han present an algorithm for mining frequent substructure patterns, which they call gSpan. It is important to note that the authors limit their discussion to frequent connected subgraphs only. gSpan explores depth-first search (DFS) in frequent subgraph mining, introducing the concepts of DFS lexicographical ordering and a minimum DFS code, which provide a canonical labeling system to support DFS searching during the mining. Definitions for these concepts can be found in [8]. The subgraph mining subroutine of the

algorithm uses a recursive graph expansion strategy, growing the current subgraph by one edge at a time to find the largest possible subgraph which is still a frequent subgraph.

2.3.5 CloseGraph

Yan and Han expand upon their work on gSpan in their paper "CloseGraph: Mining Closed Frequent Graph Patterns." They propose a new algorithm which mines only closed frequent subgraph patterns, rather than all of them. A graph g is closed if there exists no proper super-graph with the same support as g [9]. This approach addresses the issue in gSpan of the blowing up of frequent subgraphs. For example, an n-edge graph has at most 2^n subgraphs, which contribute no new information if they have the same support as the super-graph. Yan and Han postulate that their approach, CloseGraph, both drastically reduces the number of unnecessary graphs generated as well as significantly improves the efficiency of the mining[9]. Their algorithm makes use of some of the properties of gSpan, in particular DFS coding and lexicographical ordering, and the restriction of graph extensions to right-most extensions only, and defines the notion of equivalent occurence (see [9]) to provide conditions for early termination. Additionally, Yan and Han show that CloseGraph does not need to store previously discovered graphs to prune newly discovered non-closed graphs [9]. Finally, they show that CloseGraph can be used with minimal effort to mine other structures, including many other types of graphs (unlabeled, non-simple, directed, and disconnected) as well as trees.

3 Previous Work

3.1 Definitions

3.1.1 Graph Minors

A graph H is a minor of a graph G if, by performing a series of edge and vertex deletions and edge contractions on G, H can be produced from G.

3.1.2 Dependence Spheres

In their paper, "Discovering program rules and violations by mining interprocedural dependencies," Chang and Podgurski give an approach for constructing an interprocedural dependence sphere. The approach begins with the call-site graph (CSG), S, of the function of interest, f_b. Then, S is grown iteratively by adding nodes from PDGs of functions in the call tree of f_b if there exists a direct data dependency between the node and any node in S. See [2] for more details.

3.2 GMFMM

Frequent subgraph mining is an important problem in the field of specification mining, as these specifications can be modeled by graph minors of PDGs. Hence, mining the PDG for graph minors results in the discovery of program specifications. In [2], Chang and Podgurski give an approach, which they call greedy maximal frequent minor mining (GMFMM), for mining maximal frequent minors from interprocedural dependence spheres. By extending the scope of the minor to an interprocedural level, two additional types of program specifications can be learned by this approach: rules about the ordering of function calls, and rules regarding the inputs and return values of functions. The authors define the notion of maximal frequent subgraphs to be those which are not contained in any other frequent subgraphs. Note that this notion is identical to the notion of a closed graph defined in [9]. Rather than mining all frequent subgraphs, the approach looks only for frequent minors, which allows for certain variations in the structure of certain rules can be encompassed by a single rule. Since Chang and Podgurski focus on the ordering and conditional rules unique to interprocedural analysis, they offer a method to reduce the intereprocedural dependence sphere to include only call-site graph nodes and control point nodes. It is not difficult to extend the idea behind this reduction to limit the scope of the dependence sphere to any subset of node types that are of interest to the user in order to improve the efficiency of the mining algorithm.

4 Tools

4.1 Soot

Soot is a Java optimization framework, intended to be used as either a stand-alone tool to inspect class files, or to develop optimizations or transformations on Java byte code. Soot provides several different intermediate representations of Java byte code:

- **Baf**: a streamlined representation of bytecode which is simple to manipulate.
- **Jimple**: a typed 3-address intermediate representation suitable for optimization. This is the primary representation used by Soot.
- **Shimple**: an SSA variation of Jimple.
- **Grimp**: an aggregated version of Jimple suitable for decompilation and code inspection.
- **Dava**: an abstract syntax tree-based representation. Produced via decompilation of the Jimple representation.

The Soot framework is often used for its support for implementing intraprocedural data-flow analyses. Soot also provides call graph information for interprocedural analysis as part of its output.

4.2 JPDG

JPDG is a tool developed by Tim Henderson which utilizes the Soot framework to generate procedure dependence graphs (PDGs) for Java programs. The PDGs can then be mined for frequent subgraphs to infer programming rules from source code.

4.3 Parsemis

ParSeMiS is the parallel and sequential mining suite from the Friedrich-Alexander Universität Erlangen-Nüremburg. It utilizes parallel or specialized algorithms or heuristics to search for frequent, interesting substructures in graph datasets.

4.4 Python Graph Tools

graph-tool is a Python module for manipulation and statistical analysis of graphs. Many of its data structures and algorithms are implemented in C++, making it comparable in performance to a pure C/C++ graph library. The module contains a wide array of features, most notably a function to find subgraph isomorphisms.

5 Data Source

Due to the large computation time for even moderately sized projects, a test data set of small toy-programs was chosen. Ten sections of example programs were chosen from `http://www.tutorialspoint.com/javaexamples/`, a website that hosts example java code showing how to use standard library features. See Table 1 for a list of sections used a a project corpus.

	Topic
1	Arrays
2	Collections
3	Data Structures
4	Date Time
5	Directories
6	Files
7	Methods
8	Networking
9	Strings
10	Threading

Table 1: List of sections used as project corpus

For each topic, there are 8-15 examples. For use in this project, this set of files is the "program" or "project" (e.g. the Array program is a collection of 10 examples showing how to use java arrays). The hope was that programs exercising the standard library would be small enough to be computationally feasible for this project, while still having some number of non-trivial patterns.

6 Methodology and Evaluation

The following is an outline of the methodology we will use:

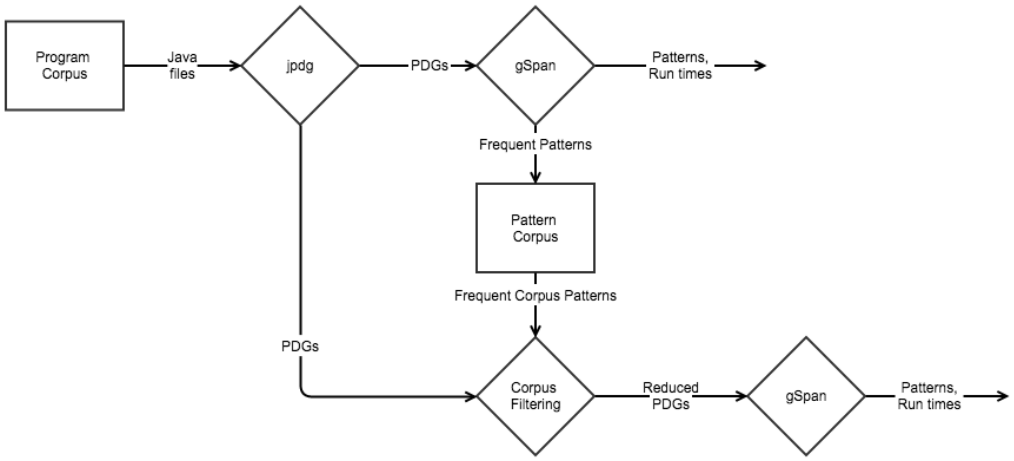

Figure 3: Flowchart of methodology used

1. Obtain a corpus of java programs
2. For each java program:
   (a) Generate the program's PDG via jpdg
   (b) Run gSpan on the PDG to discover all patterns at a given support
   (c) Record the patterns and run time of gSpan
3. Create a pattern corpus by taking the union of each program's patterns
4. For each pattern in pattern corpus:
   (a) Record the percentage of programs for which this was a frequent pattern
5. For each java program:
   (a) Generate the program's PDG via jpdg

(b) For each pattern with a corpus percentage above some threshold, remove disconnected subgraph isomorphisms from the PDG

(c) Run gSpan on the resulting PDG to discover all patterns at a given support

(d) Remove any patterns with corpus percentage above some threshold

(e) Record the patterns and run time of gSpan

7 Results

7.1 Generating PDGs

Running the programs through jpdg produces Program Dependency Graphs as dot files. Since these are at the bytecode level, the PDGS are extremely large even for small projects. Figure 4 and Figure 5 are examples of two PDGs, each generated from one file inside their respective projects. They are just shown to give a sense of scale, since they are too large for the text to be readable.

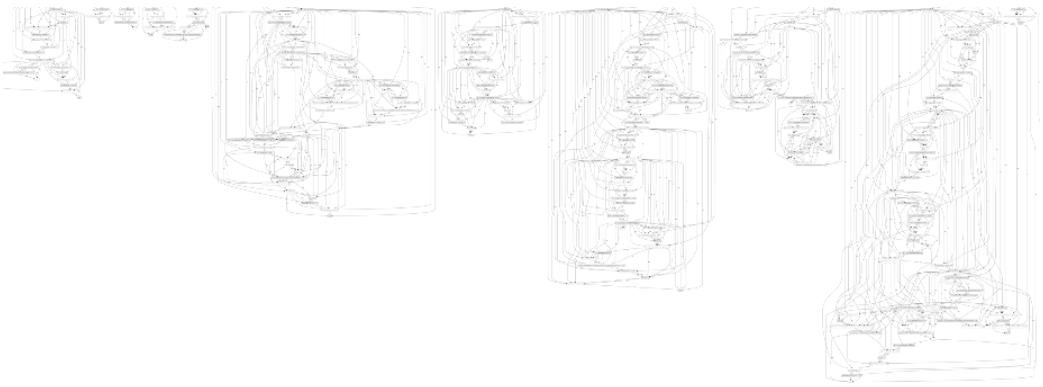

Figure 4: PDG for Reader.java, a file in the File project

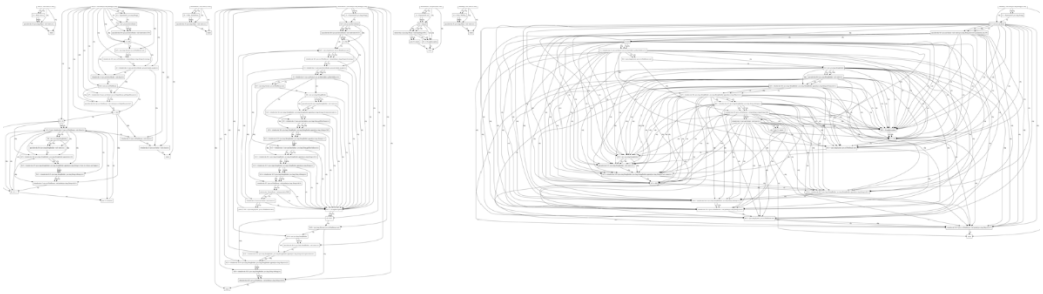

Figure 5: PDG for Server.java, a file in the Networking project

7.2 Mining Non-trivial Patterns

A "Non-trivial" pattern is one that represents an actual specification, rather than boilerplate java bytecode that will be included in most Java projects. One of the examples used in previous specification mining literature is discovering the pattern of opening and closing sockets in network programming. We therefore expect that gSpan should be able to identify a pattern such as this in the Network project. Observing the patterns found in the Network program, we see that this is the case.

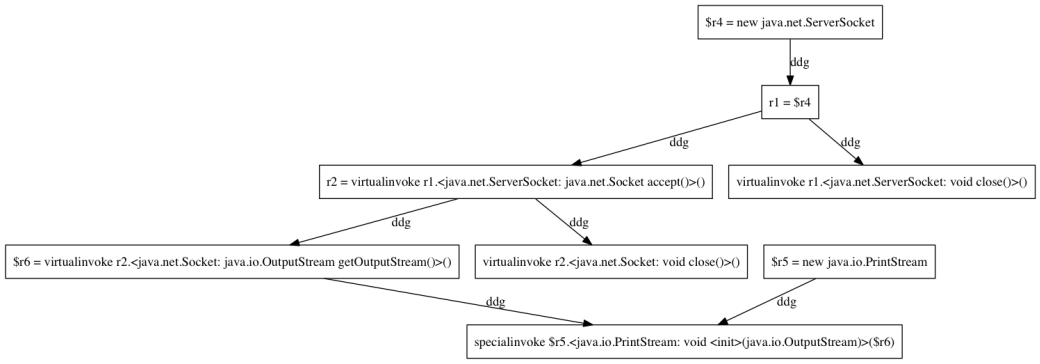

Figure 6: Frequent graph found in the Network project representing the socket open/close pattern

Revisiting the PDG for Server.java, we can mark the socket pattern by doing a subgraph isomorphism search.

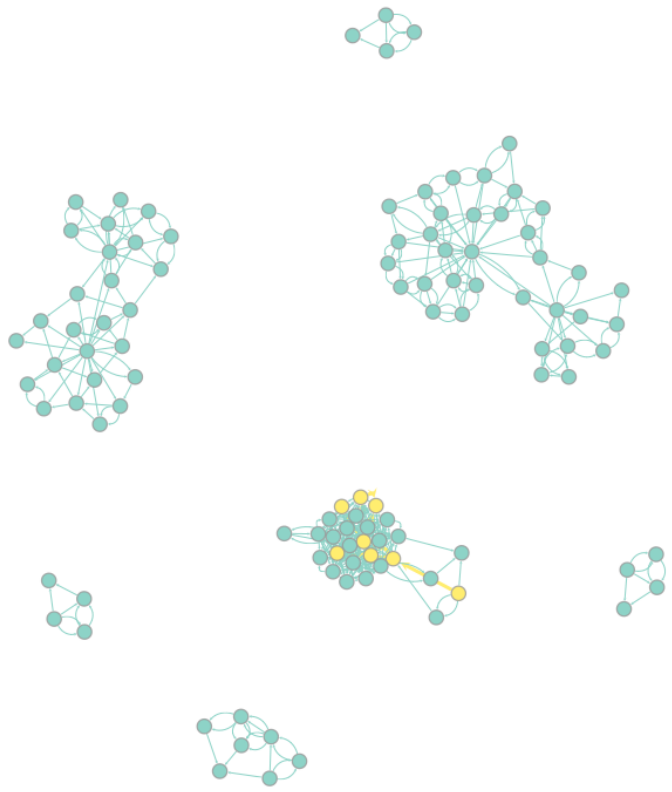

Figure 7: Highlighting socket pattern in Server.java PDG (labels omitted)

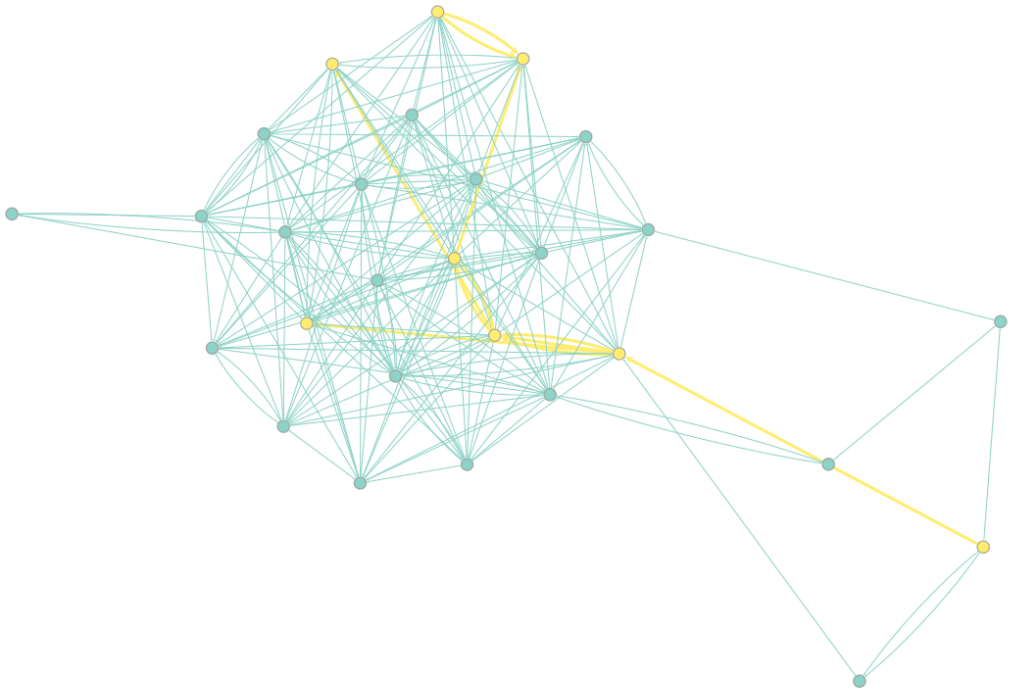

Figure 8: Zooming in on highlighted socket pattern in Server.java PDG (labels omitted)

7.3 Mining Trivial Patterns

There are many patterns that occur across a large percentage of our program corpus. These patterns are "trivial" in that they don't represent a program specification; rather, they represent common boilerplate Java bytecode.

In our test corpus, there were two patterns that were frequent patterns in over 50% of projects. Looking at these patterns, we can clearly see that they represent a getter and setter.

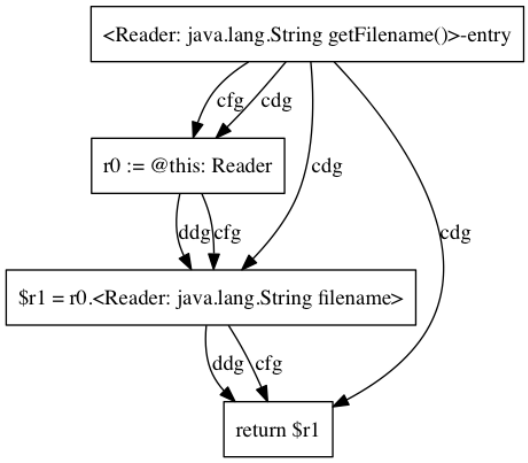

Figure 9: Getter pattern

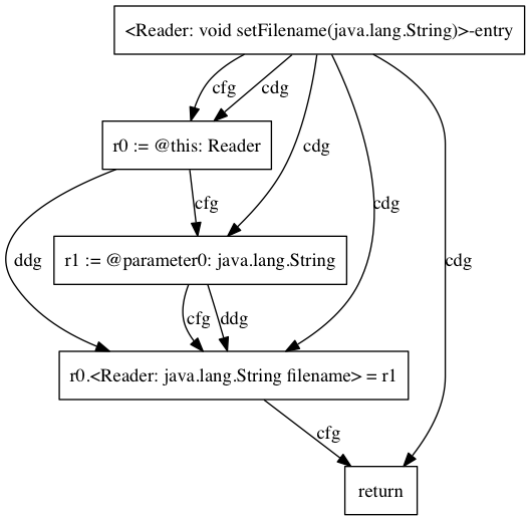

Figure 10: Setter pattern

7.4 Removing Trivial Patterns

Revisiting the PDG for Reader.java, we can mark the getter and setter by doing a subgraph isomorphism search for each.

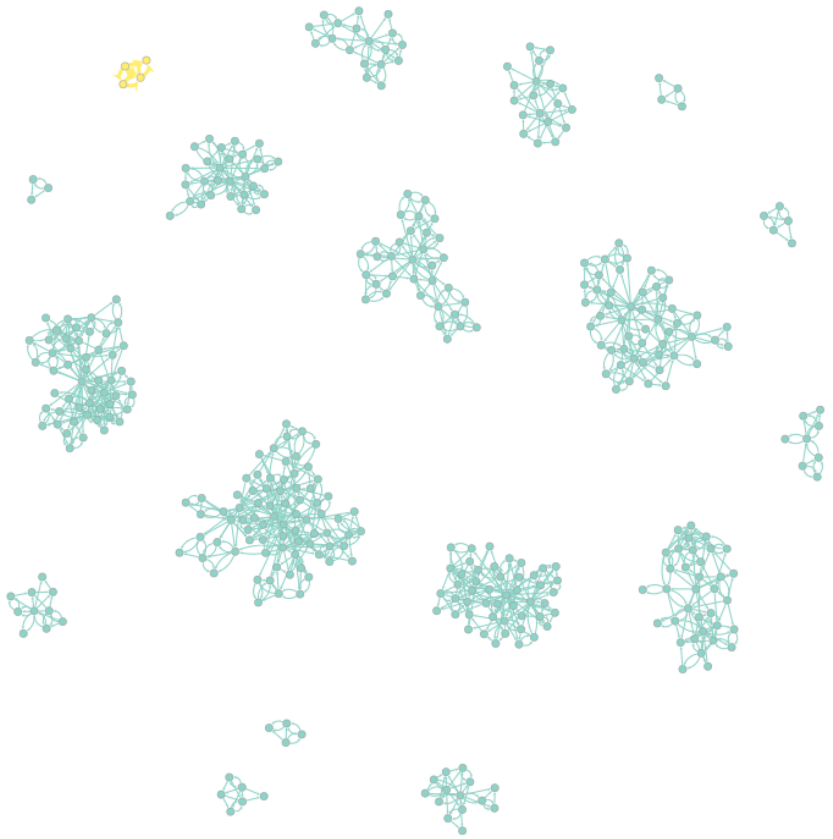

Figure 11: Highlighting getter pattern in Reader.java PDG (labels omitted)

Given that we now know what getter/setter patterns look like in java bytecode, we can filter them out from our pattern sets to reduce the number of trivial patterns we return. In fact, since these should always appear as disconnected subgraphs (unlike the socket pattern), we can remove them from future PDGs *before* running gSpan, thereby reducing our input size and reducing the number of trivial patterns.

7.5 Analysis

The methodology described in Section 6 was carried out to generate patterns with and without trivial-pattern filtering on the PDGs. The results of this,

in terms of number of patterns found and run time, are shown in Table 2.

As expected, the number of patterns returned by gSpan dropped by the number of trivial-patterns that were in the PDG previously. This is strictly an improvement in terms of pattern interestingness. While the run times were lower on the whole, the margin was very small and is likely just noise. Although we would expect removing subgraphs to improve the run time, the subgraphs that were removed were much smaller than the entire PDGs, representing a small percentage change in overall graph size. We expect that with a larger program corpus, we could find larger trivial patterns that could be removed, representing substantial gains in run time.

Project Name	**Num Patterns (at %5)**	**Time to Generate (hh:mm:ss)**	**Num Patterns after Filter (at 50%)**	**Time for Reduced (hh:mm:ss)**	**Time Ratio (Reduced/Initial)**
Arrays	34	0:40:24	30	0:39:01	0.9657
Collections	51	1:11:45	46	1:10:06	0.9771
Data Structures	20	0:34:10	18	0:39:11	1.1466
DateTime	14	0:18:36	12	0:20:47	1.116
Directories	8	0:11:31	8	0:07:19	0.6353
Files	26	0:31:09	22	0:32:58	1.0582
Methods	17	0:24:51	17	0:19:56	0.8020
Networking	37	0:41:22	34	0:41:35	1.0051
Strings	18	0:26:17	17	0:20:53	0.7943
Threading	43	1:06:26	37	1:03:39	0.9579

Table 2: Summary of pattern and run time results

8 Conclusion

In this paper we have shown that corpus-based filtering can increase the interestingness of the specifications returned from our frequent subgraph miner (using gSpan + CloseGraph), while offering a potential speedup. Future work would be to use a larger, more realistic corpus of programs on hardware/architecture that could handle the large memory requirements of doing frequent subgraph mining on java bytecode PDGs. We hypothesize that a larger corpus would allow for significant filtering before running gSpan (via subgraph isomorphism matching), thereby substantially reducing run times.

References

[1] Ammons, G., Bodík, R., and Larus, J. R. Mining specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2002), POPL '02, ACM, pp. 4–16.

[2] Chang, R., and Podgurski, A. Discovering programming rules and violations by mining interprocedural dependences. *Journal of Software Maintenance 24*, 1 (2012), 51–66.

[3] Damiani, E., Ardagna, C. A., and El Ioini, N. *Formal methods for software verification.* Springer US, Boston, MA, 2009, pp. 1–26.

[4] Ferrante, J., Ottenstein, K. J., and Warren, J. D. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst. 9*, 3 (July 1987), 319–349.

[5] Fillitre, J.-C. Deductive software verification. *International Journal on Software Tools for Technology Transfer 13*, 5 (2011), 397–403.

[6] Goues, C., and Weimer, W. Specification mining with few false positives. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,* (Berlin, Heidelberg, 2009), TACAS '09, Springer-Verlag, pp. 292–306.

[7] Lo, D., and cheng Khoo, S. Software specification discovery: A new data mining approach.

[8] Yan, X., and Han, J. gspan: Graph-based substructure pattern mining. In *Proceedings of the 2002 IEEE International Conference on Data Mining* (Washington, DC, USA, 2002), ICDM '02, IEEE Computer Society, pp. 721–.

[9] Yan, X., and Han, J. Closegraph: Mining closed frequent graph patterns. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, NY, USA, 2003), KDD '03, ACM, pp. 286–295.