Graph Mining Algorithms for Memory Leak Diagnosis and Biological Database Clustering

Evan K. Maxwell

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science
(Bioinformatics option)

Naren Ramakrishnan, Chair
Lenwood S. Heath
Godmar Back

May 10, 2010
Blacksburg, Virginia

Keywords: Graph Mining, Graph Clustering, Multipartite Cliques,
Memory Leak Detection, Bioinformatics

Graph Mining Algorithms for Memory Leak Diagnosis and Biological Database Clustering

Evan K. Maxwell

(ABSTRACT)

Large graph-based datasets are common to many applications because of the additional structure provided to data by graphs. Patterns extracted from graphs must adhere to these structural properties, making them a more complex class of patterns to identify. The role of graph mining is to efficiently extract these patterns and quantify their significance. In this thesis, we focus on two application domains and demonstrate the design of graph mining algorithms in these domains.

First, we investigate the use of graph grammar mining as a tool for diagnosing potential memory leaks from Java heap dumps. Memory leaks occur when memory that is no longer in use fails to be reclaimed, resulting in significant slowdowns, exhaustion of available storage, and eventually application crashes. Analyzing the heap dump of a program is a common strategy used in memory leak diagnosis, but our work is the first to employ a graph mining approach to the problem. Memory leaks accumulate in the heap as classes of subgraphs and the allocation paths from which they emanate can be explored to contextualize the leak source. We show that it suffices to mine the dominator tree of the heap dump, which is significantly smaller than the underlying graph. We demonstrate several synthetic as well as real-world examples of heap dumps for which our approach provides more insight into the problem than state-of-the-art tools such as Eclipse's MAT.

Second, we study the problem of multipartite graph clustering as an approach to database summarization on an integrated biological database. Construction of such databases has become a common theme in biological research, where heterogeneous data is consolidated into a single, centralized repository that provides a structured forum for data analysis. We present an efficient approximation algorithm for identifying clusters that form multipartite cliques spanning multiple database tables. We show that our algorithm computes a lossless compression of the database by summarizing it into a reduced set of biologically meaningful clusters. Our algorithm is applied to data from *Caenorhabditis elegans*, but we note its applicability to general relational databases.

Acknowledgments

This work was completed in close collaboration with my advisor, Dr. Naren Ramakrishnan, and committee members, Dr. Godmar Back and Dr. Lenwood S. Heath, all of whom have played instrumental roles in its success. I would like to thank Dr. Back for providing expertise and domain knowledge in regards to memory leaks, the Java runtime environment, and other software engineering concepts as well as for his significant contributions involving the joint work and composition of the content based in Chapter 3. I would like to thank Dr. Heath for offering his continued direction and support throughout the lifetime of this work and supplying valuable advice, especially in regards to algorithmic design and development. I am particularly grateful of the substantial guidance and oversight provided by Dr. Ramakrishnan in all aspects of this work. Finally I would like to thank Jongsoo Park, the developer of the dominator tree algorithm from the C++ Boost Libraries, for his ready responses to our questions and comments. We gratefully acknowledge support from US NSF grants CCF-0937133, ITR-0428344, and the Institute for Critical Technology and Applied Science (ICTAS) at Virginia Tech.

Contents

1 Introduction **1**

2 Background **3**

2.1 Graph Mining 3

2.2 Graph Grammars 4

2.2.1 Generators and Parsers 7

2.2.2 Graph Grammar Inference 8

2.3 Dominator Tree 13

2.4 Multipartite Graphs 14

2.4.1 Multipartite Cliques 15

2.4.2 Multipartite Clique Cover 16

2.5 Applications 16

2.5.1 Memory Leak Diagnosis 17

2.5.2 Biological Database Clustering 19

3 Diagnosing Memory Leaks **21**

3.1 Anatomy of a Leak 21

3.2 Heap Dump Processing 24

3.2.1 Generating a Heap Dump 24

3.2.2 Mining the Dominator Tree 24

3.3 Extracting Leak Candidates 25

3.4 Evaluation 28

3.4.1 Synthetic Examples 28
3.4.2 Web Application Heapdumps 30
3.4.3 Scalability and Other Quantitative Aspects 34
3.5 Related Work 37
3.5.1 Memory Leak Detection Tools 37
3.5.2 Data Mining for Software Engineering 38

4 Clustering Biological Data **39**
4.1 Computational Models for Gene Silencing Database 39
4.1.1 Background 39
4.1.2 Graph Structure and Generation 40
4.1.3 Clusters of Multipartite Cliques 42
4.2 Approximate Algorithm for Multipartite Clique Cover 44
4.2.1 Method 44
4.2.2 Variations 48
4.3 Results 50
4.3.1 Seed Selection 54
4.3.2 Closure of Multipartite Cliques 57
4.3.3 Biological Analysis of Clusters 57
4.4 Related Work 62

5 Conclusions **64**

Bibliography **65**

List of Figures

2.1 Examples of context free and node-label controlled graph grammars. 6
2.2 Examples of variables and recursion in a graph grammar. 11
2.3 (i) An example graph and (ii) its dominator tree. 14
2.4 Bipartite graph and bicliques/biclusters. 15

3.1 An example of a leak "hidden" underneath legitimate objects. 22
3.2 Heap graph after executing the program shown in Figure 3.1. 23
3.3 Examples of subgraph frequency variations between graph and dominator tree. 26
3.4 Examples of graph grammar mining on dominator trees. 27
3.5 Most frequent grammar mined from HiddenLeak example in Figure 3.2. . . . 28
3.6 Root path grammar of TreeMap container for Figure 3.2. 30
3.7 A singly linked list embedded in an application class. 31
3.8 Most frequent grammar in synthetic linked list example. 32
3.9 Most frequent grammar in Tomcat 5.5 heap dump. 33
3.10 Most frequent grammar in MVEL heap dump. 34
3.11 Runtime statistics on real heap dumps. 36

4.1 Adjacency information for CMGS multipartite graph. 42
4.2 Overlapping multipartite cliques. 44
4.3 Example of multipartite clique discovery algorithm. 49
4.4 Example of strict $1 \times N$ biclique checking. 51
4.5 Number and distribution of clusters returned with different variations. 52

4.6 Clustering edge coverage results. 53
4.7 Clustering distribution with different seed node selection. 55
4.8 Edge coverage results with different seed node selection. 56
4.9 Clustering edge coverage histogram with different seed node selection. 58
4.10 Distribution of biclique sizes from clustering. 59

List of Tables

2.1 Graph grammar terminology. 5

3.1 Heap dump summary statistics. 35

4.1 Node and edge distribution of CMGS multipartite graph. 43

4.2 Algorithm terminology. 45

Chapter 1

Introduction

Graph mining is an area of research aimed at discovering knowledge from large graph-based datasets in the form of descriptive subgraph components. Graph mining is a general term, as the breadth and depth of graph mining has expanded to many application domains and incorporated vastly dissimilar techniques. Such applications include biology, chemistry, software engineering, social networks, web data, image processing, etc. Graphs are quite common because of their modelling capabilities; seemingly unstructured datasets can often be implicitly modelled as a graph. The benefit of this is the plethora of algorithms and overall techniques that computer scientists have designed specifically for efficiently and effectively analyzing graphs. Graph mining encompasses the overlap between graph theory and data mining, which alone are two integral areas of computer science research.

The motivation for our work in graph mining follows a central research methodology, which can be summarized into five stages:

1. Characterize the problem domain.
2. Identify a class of graph patterns that are valuable within that domain.
3. Define quality metrics for assessing the patterns.
4. Design efficient algorithms to extract the patterns.
5. Apply the method to real and synthetic datasets.

This research methodology allows us to take a directed approach to solving real problems with graph mining.

In this thesis, we apply our research methodology to two applications. First, we use graph mining as the basis of a debugging tool for diagnosing memory leaks in Java programs. We generate a graph dataset by dumping the heap of a Java program and construct a single,

directed graph based on object ownership relationships. From this graph, we aim to identify frequently occurring graph patterns that indicate sets of objects that consume a large portion of heap memory. We anticipate that these frequent subgraphs will correspond to potentially leaking objects which we can use to contextualize possible memory leaks. In Chapter 2, we describe more background information about memory leaks as well as introduce the quality metrics and algorithms we use to efficiently extract them. Chapter 3 presents the details of our algorithm and the results of running it on both real and synthetic datasets.

In our second application, we analyze a heterogeneous biological database for *Caenorhabditis elegans* with graph mining to provide a summarization of the contents of the database, as well as to extract informative components hidden within the expanse of data. Such biological databases contain massive amounts of data integrated from multiple sources and the potential insight this data can provide to biologists is far beyond what can be extracted with manual analysis. We design a graph clustering method to decompose the database into subgraph clusters of interrelated biological entities to present an overview of the contents of the dataset. In Chapter 2, we provide more information about the importance of these types of databases in biological research and detail the subgraph clusters we aim to extract along with their evaluation scheme. In Chapter 4 we explain the design of our mining algorithm and present its performance on the *C. elegans* database. Finally, Chapter 5 concludes with a summary of the contributions of our work, a discussion of the implications, and a description of future directions for research in regards to both applications.

Chapter 2

Background

In this chapter, we introduce the concepts, methodologies, and application details that we consider background information for our work, followed by a brief statement of what our contributions are to the field.

2.1 Graph Mining

Initial works by Kuramochi, et al. [30] and Yan, et al. [58] present traditional frequent subgraph mining algorithms, which are similar to frequent itemset mining or closed itemset mining [60] with the added constraints imposed by the structure of the graph. In itemset mining, the input data is a set of transactions containing sets of items, and the goal is to extract all subsets of items that occur together frequently in the set of transactions. Frequent subgraph mining can be naturally formulated as an extension of frequent itemset mining with the added computational complexity of recognizing isomorphism between two graphs. In frequent subgraph mining, the data is a set of graphs (transactions) containing sets of nodes (items) and edges represent distinct relationships between nodes. Subgraph isomorphism (is a graph G_1 a subgraph of G_2) is known to be an NP-complete problem, and graph isomorphism (does there exist a bijection between nodes in graph G_1 onto the nodes in graph G_2) is known to be in NP but is not known to be in P nor NP-complete [19]. Thus frequent subgraph mining is more computationally complex than frequent itemset mining. Efficiently generating and evaluating candidate subgraphs is a well studied subproblem of graph mining. For example, in the gSpan algorithm presented by Yan, et al. [58], subgraph isomorphism is approached by creating a canonical form for subgraphs based on depth-first search so that isomorphism tests can be avoided.

The added complexity in identifying frequent subgraphs motivates the search for efficient algorithms tailored towards specific applications. In many cases, efficiency gains are achieved by exploiting properties of the desired subgraph patterns as well as the input graph(s).

Examples include the expansion of the FP-tree data structure [20] for use in fixed-structure graphs [4] and exploiting edge-cut techniques when searching for cliques [61]. In other cases, the desired graph patterns and application itself may make the discovery process more computationally complex, in which case efficient mining strategies have added importance. Examples include subgraphs with geometric coordinates in [31], requiring that orientation flexibility and fuzziness be allowed in candidate structures, and inexact subgraph pattern matching in [11].

In graph mining, the goal is to extract knowledge from a large graph or set of graphs in an effort to comprehend its key features. The knowledge component can take various forms and many methods have been proposed for graph mining. In this work, we study graph mining techniques through the application of our research methodology on two problems — memory leak diagnosis and biological database summarization.

2.2 Graph Grammars

In this section we introduce the concept of graph grammars. Graph grammars are similar in concept to formal language grammars in that the goal is to define the basic concepts of the language with a set of low level rules on how sentences of the language can be constructed. With graph grammars, the sentence is a connected graph. Specifically, we are interested in node-label controlled (NLC) context free graph grammars as defined in the graph grammar mining literature [27]. Table 2.1 presents these definitions.

We note that these definitions for NLC and context free differ from those of the broader graph grammar literature, particularly [16]. In the broader literature, NLC graph grammars are defined as a class of graph grammars that attempt to mimic the recursive properties of traditional context free grammars in graphs. They are not, however, context free in the sense that derivations are independent of the order in which productions are applied. NLC graph grammars contain connection instructions controlling how to reconnect incident edges during the embedding process using node type labels, thus ensuring that the graph transformation is completely local and making them more free of context. In the graph grammar mining literature, NLC means only that the left-hand side of the production rule is a single node and context free implies production rules may be applied devoid of restrictions regarding the neighborhood. Figure 2.1 exemplifies the characteristics of the NLC context free graph grammars that we use from the graph grammar mining literature. Notice that because the grammars are context free, there is no canonical way to reconnect the neighborhoods of the non-terminal nodes after embedding via productions S_A and S_B. The importance of looking specifically for NLC context free graph grammars lies in their ease of use for graph mining and will become clear later.

By maintaining NLC, the grammar ensures a one-to-many relationship between the non-terminal and terminal nodes on either side of a production and therefore guarantees that

Table 2.1: Graph grammar terminology.

Graph grammar	A tuple $(\Sigma, N, \mathbb{P})$, where Σ is a set of terminal labels, N is a set of non-terminal labels, and $\mathbb{P}$ is a set of production rules.
Terminal	A label that can not be replaced. Terminals define unique, primitive labels of the graph language. Terminals can only occur on the right-hand side of a production rule.
Non-terminal	A label that can be replaced. Non-terminals may occur on both the left-hand and right-hand side of a production rule. Thus, a non-terminal can be replaced with a set of terminals, non-terminals, or a combination of both via a production rule.
Production rule	A rule of the form $S \rightarrow P$ where S may contain only non-terminals and P is a set of subgraphs that may contain terminal and/or non-terminals. This rule implies that any instance of S may be replaced by a subgraph defined in the set P.
Embedding	The process of applying a production rule by replacing an instance of S with a subgraph defined in P and reconnecting the neighborhood of S (nodes and edges $\notin S$) to the new subgraph from P.
Context free	Implies that the context of S is not considered when a production rule $S \rightarrow P$ is applied. Thus it is always legal to perform a production on S, regardless of the structure of the neighborhood of S.
Node-label control (NLC)	Restricts the left-hand side (S) of a production rule to a single, non-terminal node.

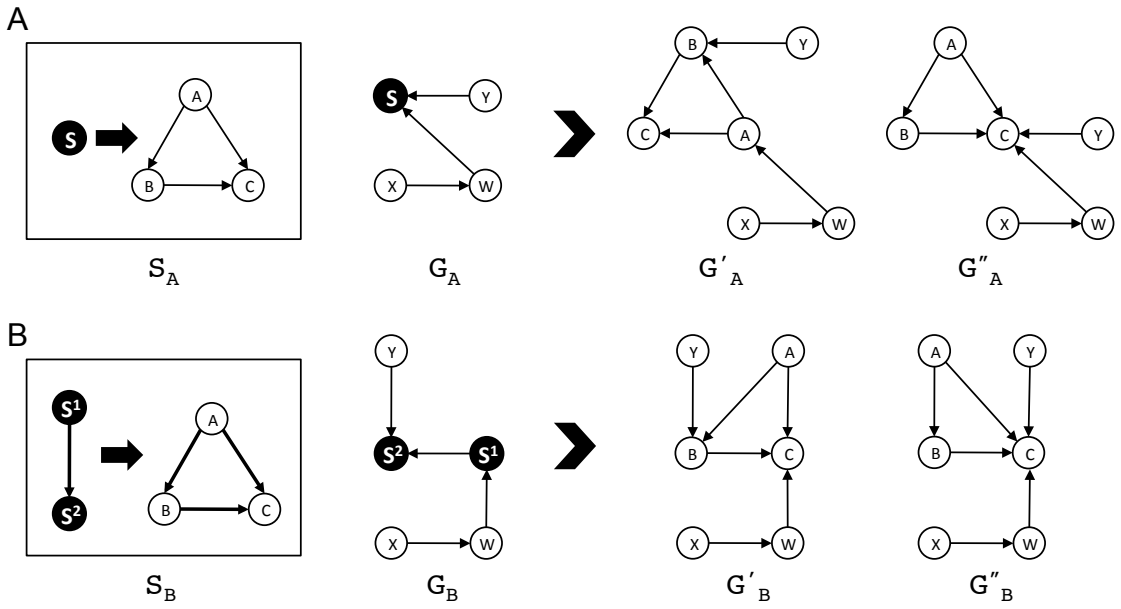


Figure 2.1: (A) An example NLC context free graph grammar production rule S_A is applied to a graph G_A. $G_A^{'}$ and $G_A^{''}$ show two ways in which the neighborhood of the non-terminal node S can be restructured following the embedding of production S_A in G_A. This flexibility is a result of context freeness. (B) An example graph grammar production rule S_B is applied to a graph G_B. Like S_A, S_B is context free, but it is not node-label controlled because the left-hand side of the production $S \rightarrow P$ is not a single node. Again, $G_B^{'}$ and $G_B^{''}$ show how the context freeness allows for flexibility in reconstructing the neighborhood after embedding.

executing a production rule will have a strictly non-decreasing effect on the size of the graph. By maintaining context freeness, a graph grammar becomes more space efficient because it does not need to keep track of the neighborhood of a given production instance. A production rule $(S \rightarrow P) \in \mathbb{P}$ can contain many graphs on the right-hand side (recall that P is a set), each of which may contain, in addition to terminal labels from Σ, more non-terminal labels from N including S. These characteristics allow for graph grammars to exhibit two important behaviors. First, they can be recursive and allow production rules of the form $S \rightarrow Sx$, where $x \in \Sigma$ and $S \in N$, such that a non-terminal can create more non-terminals of the same type. Second, the right-hand side P of a production rule defines a set of subgraphs, thus variability is created by allowing multiple patterns. Variable production rules take the form of $S \rightarrow A \mid B \mid C$ so that an instance of S could be replaced with an instance of A, B, or C. A NLC context free graph grammar has many applications but is generally characterized by its use in either graph generation or graph parsing [27].

2.2.1 Generators and Parsers

As described previously, the format of a production rule in a NLC context free graph grammar is $S \rightarrow P$ where S is the non-terminal node and P describes the subgraphs that can be used to replace S. This type of grammar can be used in multiple ways, namely, for generation and for parsing. In generation, a graph grammar is used to generate a graph or set of graphs that fit the language defined by the production rules of the grammar. In this case, the idea is to use the grammar to build example "sentences" of the language. In graph grammar parsing, the opposite is true. The input is an example graph or set of graphs that are elements of a specific language. Using the example graph, the graph grammar is used to systematically decompose the graph based on the grammatical rules expressed by the language.

Generator graph grammars are useful for building sets of graphs that exhibit similar properties and may be used for building synthetic datasets. Parser graph grammars, on the other hand, are useful for trying to break down a graph and determine the sequence of production events that were used to create it. Context free graph grammars provide efficiency for both generators and parsers, but they are also lossy. However, when the graph grammar exhibits NLC as well, the lossiness affects generators more than parsers. Since the left-hand side of the production rule is a single node, reconnecting the neighborhood after a parse is straightforward. For a generator, however, even a trivial case such as Figure 2.1(A) where a single production takes place on G_A, without context, embedding ambiguously produces G'_A or G''_A. Parsing G'_A and G''_A with S_A always produces a graph isomorphic to G_A. Parsing with a NLC context free graph grammar can still be ambiguous, but the ambiguity is not a result of reconnecting the neighborhood. The lossiness due to being context free implies that once a parse takes place, we no longer know the context of the original graph and can not faithfully reproduce it.

2.2.2 Graph Grammar Inference

For our application, we are interested in the inference of graph grammars from a large graph dataset. In this case, the grammar is used for parsing the graph but its production rules are unknown *a priori*. This task resembles an unsupervised learning problem and the main complexity lies in efficiently finding the most descriptive production rules without making them too complex. The algorithm that we base our method on is presented in [27, 29]. This algorithm is an extension of `SUBDUE` [13], which takes a slightly different approach to graph mining. In this work, instead of reporting all subgraphs that occur above some threshold, the input graph is viewed as a set of information that takes a certain number of bits to describe. `SUBDUE` uses the Minimum Description Length (MDL) principle from [47] as a scoring function to find a single, most descriptive subgraph. The extensions in [27, 29] use this same principle, but look for the most descriptive production rules to build a graph grammar. In this case, the best production rule inferred from a large graph dataset is that which when parsed out of the input graph causes the largest reduction in the MDL for representing the transformed graph as well as the rule used to parse it.

This differs slightly from traditional graph grammar parsing as described in Section 2.2.1, because we do not parse instances of the productions one at a time. Instead, the *a priori* candidate generation strategy builds a set of instances matching the right-hand side of the production rule, some of which may be overlapping and possible sources for ambiguous parsing. However, we do not care about the exact parsing hierarchies that could be used to perform the transformation; rather, we care only about the end result where all of the nodes and edges contained in the set of instances are removed and replaced with a minimal set of single non-terminal nodes labeled with the left-hand side of the production rule. The basics of the algorithm from [29] are presented in Algorithm 1, where the graph grammar is built one production rule at a time, and the beam width β corresponds to the maximum size of a priority queue Q containing top-scoring candidate production rules. The RecursifySubstructures function checks and updates a candidate production rule for recursion. Note we say that we *compress* G on the best production rule `bestSub` indicating a one-time, global parsing transformation.

This method has multiple advantages over traditional graph mining techniques that simply take an input threshold argument and return all subgraphs that are frequent enough to meet the threshold. First, it does not require that the user specify an input threshold, thereby eliminating the uncertainty associated with choosing a good value. Furthermore, the output from the algorithm is a graph grammar containing a single, highly descriptive production rule or, if run iteratively, a ranked set of descriptive production rules. Other algorithms generally return a set of subgraphs, which may contain subsets of each other, and their absolute frequencies. Second, using graph grammar productions as opposed to discrete subgraphs allows for similar but non-isomorphic graphs to be generalized to a single pattern. This allows for flexibility in the graph grammar and presents a tradeoff between the specificity of the grammar and the amount of the input data it can represent. This

Algorithm 1 InferGraphGrammar(G, β):

Input: An input graph G and a beam width β.
Output: The best graph grammar, a set of productions $\mathbb{S}$.

$\mathbb{S} \leftarrow \emptyset$
repeat
  $Q \leftarrow \{\beta$ most frequent node types $v \in G\}$
  `bestSub` $\leftarrow$ first $v \in Q$
  repeat
    $Q^{'} \leftarrow \emptyset$
    for each $S \in Q$ **do**
      `newSubs` $\leftarrow$ all single edge extensions of S
      `newSubs` $\leftarrow$ RecursifySubstructures(`newSubs`)
      $Q^{'} \leftarrow$ `newSubs`
      evaluate all $S \in Q^{'}$ with MDL, keeping only the top β
    end for
    `bestSub` $\leftarrow \underset{S}{\text{argmax}} \{$`bestSub` $\cup$ `newSubs`$\}$
    $Q \leftarrow Q^{'}$
  until Q is empty
  $\mathbb{S} \leftarrow \mathbb{S} \cup$ `bestSub`
  $G \leftarrow G$ compressed on `bestSub`
until `bestSub` cannot compress G
return $\mathbb{S}$

tradeoff is representative of Occam's Razor, the best model is generally the simpler model, and the MDL principle takes this into consideration. Finally, graph grammars can contain recursive constructs such that the left-hand side of a production rule can also be contained in the right-hand side. In practice, we generalize this to mean that the right-hand side P of a production can include connection instructions such that specific nodes, edges, or subgraphs contained in P may be used to connect to another production of P.

The inclusion of recursive production rules has an important influence on the inference algorithm. When a production rule contains recursion, then its instances may be overlapping in the input graph. Recall that when we compress a graph with a production rule, then any instance of a subgraph in the right-hand side P of the production will be transformed into a single non-terminal node labeled with the left-hand side S. If the production is recursive, then multiple overlapping instances will become a single non-terminal node, not a set of non-terminal nodes for each unique instance. When calculating the score of the production, this additional node reduction must be accounted for. Specifically, if a set of n instances of a production are connected, then the resulting compressed graph will have $n - 1$ fewer nodes than the compressed graph that fails to identify the recursion. Figure 2.2 shows how recursion affects compression and also how non-specific pattern matching (now on referred to as variables) can occur in a graph grammar. The production rule in B also exemplifies the notation of productions from our NLC context free graph grammars. Variables on node or edge labels imply a production rule whose right-hand side is a set of unique terminal labels. Thus we simplify productions of the form:

$$S_1 \rightarrow (A \leftarrow S_2)$$
$$S_2 \rightarrow B \mid C \mid D$$

to a single production of the form:

$$S \rightarrow (A \leftarrow \{B \mid C \mid D\})$$

This substitution simplifies the mining process. To indicate a recursive production, we use the (S) notation and a gray box to indicate the node or edge replacement rules of the production S [29]. We also allow productions to have *complex replacement* rules, where the recursion can happen over a subgraph larger than just a single node or edge (see an example in Figure 3.9). Note the "$G|S$" notation implies "G given S", which is the graph G compressed on production rule S.

The additional expressiveness of graph grammars is a favorable characteristic for extracting knowledge from the input graph, but it does come with additional computational costs in regards to both time and space. For each of the three advantages mentioned above, we add complexity to the traditional *a priori*, level-wise candidate generation and evaluation steps.

Candidate Evaluation

In traditional graph mining, candidates are evaluated simply by counting the number

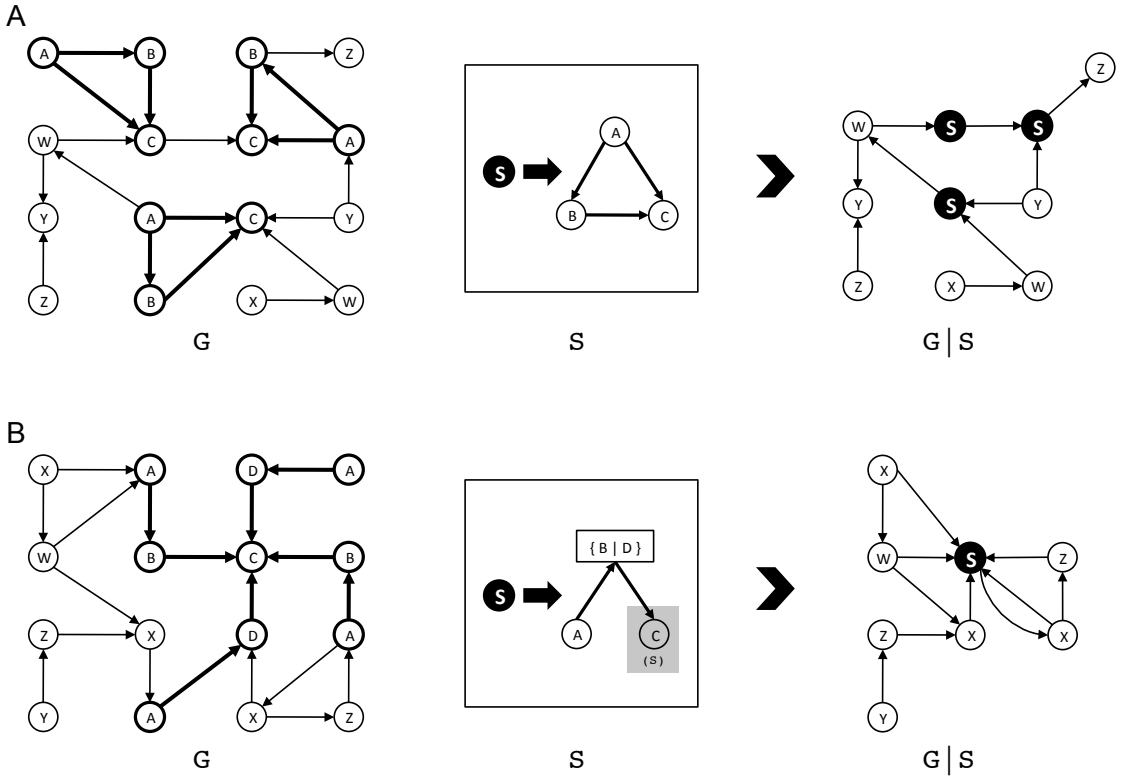


Figure 2.2: (A) An example graph grammar S is used to parse the input graph G. In this case, S defines a non-recursive graph grammar containing no variables and each instance of the production is replaced with a non-terminal node labeled S. (B) An example graph grammar that contains both variable labels $(B \mid D)$ and a recursive connection instruction (the "(S)" designation on node C). In this case, all patterns matching $A \rightarrow B \rightarrow C$ or $A \rightarrow D \rightarrow C$ are part of the grammar, and the recursion about node C implies the connected productions can be reduced to a single non-terminal node.

of instances of the candidate subgraph and checking if the occurrence count is larger than the threshold. This final evaluation comparison is a constant time operation, and can be a constant space operation if the subgraph instances do not need to be enumerated. By using the MDL scoring heuristic, each candidate subgraph must be analyzed with respect to the entire input graph, which requires enumeration of all instances of the production. In a worst-case scenario, this becomes at least linear in the size of the graph ($|V| + |E|$) in both time and space.

Variable Labels

As described previously, candidate graph grammar production rules can contain variable node and edge labels via productions of the form $S \rightarrow A \,|\, B \,|\, C$. This allows us to mimic the behavior of inexact pattern matching. A very general graph grammar could be created to cover the entire graph with a single production rule containing variables matching all edges between all nodes, i.e., $S \rightarrow (N \leftrightarrow N)$ and $N \rightarrow$ (any label $\in G$). However, this is non-descriptive and MDL would score it poorly given that N is a large set. To efficiently pick descriptive variable patterns, the algorithm first considers only high coverage non-variable patterns. After iteration k of finding the set of candidate productions S_k, the algorithm creates new productions by merging similar productions from S_k to create variable labels. This additional step is not time consuming in practice, but is an $O(|S_k|^2)$ operation where $|S_k|$ is the number of candidate productions in S_k. In addition, if all $|S_k|$ productions can be merged together, this could theoretically create $2^{|S_k|} - 1$ productions using all non-empty combinations. In practice, however, we would restrict this scenario to only $|S_k|$ possible productions by first ranking the productions in S_k by score and adding one at a time to create new productions. This adds minimally to the space complexity of the algorithm.

Recursive Productions

In order to determine if a candidate production is recursive requires checking if overlap occurs between instances of the production in its native, non-recursive form. This is an $O(|S_k| \cdot n\ log(n))$ operation, where n is the number of nodes in all instances of a production $S_{k_i} \in S_k$, and must be performed at every iteration k. Further, when recursion is identified, the MDL scoring function used in candidate evaluation must reflect this information. Keeping track of the recursion details for a candidate production does not significantly add to the space complexity of the algorithm.

Evidently, finding a graph grammar adds computational complexity to the *a priori*-style graph mining algorithm. When the algorithm is presented in [27, 29], the input graph is not large and these additional complexities are negligible. In our work, we use sampling during the pattern extension, recursion detection, and candidate scoring phases of the algorithm to improve runtime.

2.3 Dominator Tree

The dominator tree is a concept used heavily in compilers for code optimization. The dominator tree is built on the "dominates" relationship defined between nodes in a directed G. In order to explain how a dominator tree is created from G, we must first state the assumptions about G and define some terminology.

Assumptions

1. G must be a directed graph.
2. G has exactly one source node.

Terminology

1. **Entry node:** An entry node of G is a node with only outgoing edges.
2. **Dominates:** For 2 nodes $x, y \in G$, x dominates y ($x \gg y$) if and only if every path from the entry node to y passes through x. x is called a **dominator** of y so long as $x \neq y$.
3. **Immediate dominator:** For 2 nodes $x, y \in G$, x is the immediate dominator of y if and only if $x \gg y$ and there does not exist a node $z \in G$ such that $x \gg z$ and $z \gg y$.

From these definitions it follows that a node may have many dominators unless it is an entry node, in which case it will not have any, and every non-entry node is dominated by the entry node. Further, a node can have at most one immediate dominator, but that dominator may be the immediate dominator of many nodes. Under these definitions and assumptions, we define the dominator tree.

Definition 1. *The **dominator tree** $D = (V^D, E^D)$ is a tree induced from the graph $G = (V^G, E^G)$, where $V^D = V^G$, but an edge $(u \to v) \in E^D$ if and only if u is the immediate dominator of v in G.*

Computing the dominator tree efficiently is a well studied topic. We adopt the Lengauer-Tarjan [33] algorithm implementation from the Boost C++ Graph Library [50]. This algorithm runs in $O((|V| + |E|) \log(|V| + |E|))$ time, where $|V|$ is the number of vertices and $|E|$ is the number of edges. Figure 2.3 shows an example input graph and the dominator tree induced from it. In this example, node A is the entry node and does not have a dominator. The dominator set for node G is $\{A, D\}$, because all paths to G must go through these nodes, and D is the immediate dominator. In this case, these edges do not change between the graph and the dominator tree. The set of paths to node L, however, only passes through node A in all cases, so A is the only dominator, and therefore the immediate dominator of L. Thus the dominator tree contains the edge $A \to L$.

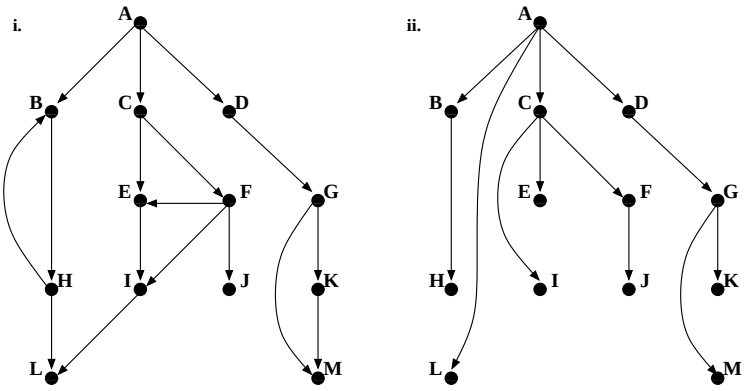


Figure 2.3: (i) An example graph and (ii) its dominator tree.

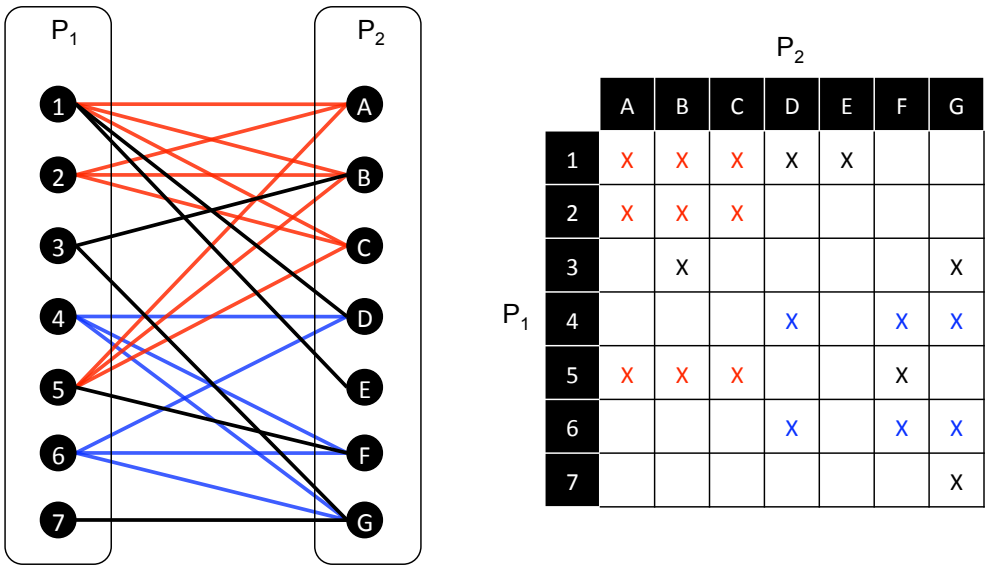
2.4 Multipartite Graphs

Multipartite graphs are a special class of labeled graphs where nodes of the same label are grouped (partitioned) into blocks and no edges occur between two nodes having the same label.

Definition 2. *A* ***multipartite graph*** $\mathcal{MG}$ *containing* n *blocks is constructed such that* $\mathcal{MG} = (V, E)$ *where* $V = \bigcup_{i=1}^{n} V_i$ *and* $E = \bigcup_{\substack{i,j \\ i \neq j}} E_{ij}$, *where* $E_{ij} \subseteq V_i \times V_j$.

Multipartite graphs are an important class of graphs because, among other reasons, they can be constructed from relational databases (RDBs) by the primary key-foreign key relationship. In this case, each block of the graph represents a table in the RDB and edges occur between nodes if there is a link between keys in the rows of the tables. Although this is true for general RDB tables, we are particularly interested in tables that have a many-to-many relationship as these imply a high degree of connectivity between two blocks.

Bipartite graphs are a well-studied subclass of multipartite graphs, where $\mathcal{MG}$ is restricted to only two blocks V_1 and V_2, and the edge set is $E \subseteq V_1 \times V_2$. Because bipartite graphs contain exactly two blocks, they can be modelled as a set of pairwise relationships and be considered a binary dataset. Because they do not contain edges between nodes of the same block, the adjacency matrix can contain only nodes of type V_1 in one dimension and V_2 in the other. This property allows a bipartite graph to be used to model many well known problems in data mining. Specifically, a bipartite graph can model a frequent itemset mining problem where rows are the transactions and columns are the items, and also a biclustering problem where biclusters are represented by bicliques. Figure 2.4 shows this transformation, where bicliques in red and blue correspond to biclusters as well as frequent sets of items identified in a set of transactions.



P_1 \ P_2	A	B	C	D	E	F	G
1	X	X	X	X	X		
2	X	X	X				
3		X					X
4				X		X	X
5	X	X	X			X	
6				X		X	X
7							X

Figure 2.4: A bipartite graph and the corresponding adjacency matrix. Highlighted in blue and red are bicliques, or biclusters.

2.4.1 Multipartite Cliques

Cliques are a class of graph patterns that are well studied in graph mining because of their importance in many application domains. A clique is a graph such that every node is connected to every other node in the clique, forming a cluster of interrelated nodes. The strong inter-connectivity amongst the nodes of a clique indicates that some grouping may exist. For example, cliques in a protein-protein interaction network often represent protein complexes [5]. A large amount of research exists on how to mine cliques efficiently, some of which relaxes the definition to include what are termed pseudo-cliques or quasi-cliques [5, 59, 61] where an edge density threshold or similar metric may be used to determine nearly complete connectivity.

When the input graph is partitioned, it is not possible to obtain a clique with more than one node per partition block since nodes within the same block can not have edges between them. In bipartite graphs, the notion of a clique is relaxed to a biclique, a set of nodes spanning across both blocks such that all nodes in opposite blocks are connected. Figure 2.4 shows two example bicliques in red and blue. We define a multipartite clique similar to a biclique.

Definition 3. *A* ***multipartite clique*** $\mathcal{C}$ *is a tuple* $(\mathcal{P}, \mathcal{V}, \Lambda)$*, where* $\mathcal{P}$ *is a set of partition blocks,* $\mathcal{V}$ *is a set of nodes in the blocks* $\mathcal{P}$*, and* Λ *is a set of block pairs containing the adjacency information of the clique.*

In fact, we consider bicliques a special subset of multipartite cliques when $|\mathcal{P}| = 2$. Once

$|\mathcal{P}| > 2$, the clique property must become slightly more relaxed. Specifically, consider a multipartite clique C with $\mathcal{P} = \{V_A, V_B, V_C\}$, $\mathcal{V} = \{V_A^C, V_B^C, V_C^C\}$, and $\Lambda = \{(V_A, V_B), (V_B, V_C)\}$. Thus nodes in V_A are never connected to nodes in V_C. This scenario is common in multipartite graphs and RDBs. We consider a multipartite clique to be a union of bicliques defined on the partition block pairs in Λ. Further, a multipartite clique defined on a multipartite graph need not cover all partition blocks in the multipartite graph. This implies in our example that the biclique $V_A^C \times V_B^C \subset C$ is a multipartite clique on its own. Other restrictions include that the adjacency information in Λ must be connected. Our definition differs slightly from [14] because we do not require an order among partition blocks. In addition, from our example for instance, we also allow adjacency between V_A and V_C making the blocks completely connected and we call this *cyclic* behavior. Multiple problems on finding maximal multipartite cliques are formulated in [14], the most similar of which is the "Multipartite Clique Problem which Includes Nodes from Some Levels" (MPCS) and is shown to be NP-complete when the edges are weighted.

2.4.2 Multipartite Clique Cover

Using our definition of a multipartite clique from Section 2.4.1, we define the problem of Multipartite Clique Cover (MCC).

Definition 4. *A **multipartite clique cover** is a set $\mathbb{C}$ of edge-disjoint multipartite cliques that cover all edges of the multipartite graph $\mathcal{MG}$.*

We define the optimization version of the multipartite clique cover problem with respect to the set cover optimization problem from [19], which is known to be NP-hard. In our case, the items of the sets are edges in a multipartite graph and the sets are multipartite cliques. The goal is therefore to pick the multipartite clique cover defining the smallest set $\mathbb{C}^*$ that covers the edges E of $\mathcal{MG}$ completely. Further, we disallow overlap between cliques in a multipartite clique cover $\mathbb{C}$, so every edge in every $\mathcal{C} \in \mathbb{C}$ is unique.

The fact that the set cover optimization problem that this mimics is NP-hard illustrates the need for an approximation algorithm. Further, MPCS from [14], although a slightly harder problem due to edge weights, is NP-complete for finding a single maximal multipartite clique. We will present an approximation algorithm for this problem in Chapter 4 and will describe its use as an approach for multipartite graph clustering. This problem definition and application to graph clustering is novel to the best of our knowledge.

2.5 Applications

The terminology defined thus far provides the necessary background information on the concepts and terminology that we refer to in the technical descriptions of our work. In

this section, we will introduce the applications that motivate our work and provide some background for each topic.

2.5.1 Memory Leak Diagnosis

Memory leaks are a frequent source of bugs in applications that use dynamic memory allocation. They occur if programmers' mistakes prevent the deallocation of memory that is no longer used. Undetected memory leaks cause slowdowns and eventually the exhaustion of all available memory, triggering out-of-memory conditions that usually lead to application crashes. These crashes significantly affect availability, particularly of long-running server applications, which is why memory leaks are one of the most frequently reported types of bugs against server frameworks.

Memory leaks are challenging to identify and debug for several reasons. First, the observed failure may be far removed from the error that caused it, requiring the use of heap analysis tools that examine the state of the reachability graph when a failure occurred. Second, real-world applications usually make heavy use of several layers of frameworks whose implementation details are unknown to the developers debugging encountered memory leaks. Often, these developers cannot distinguish whether an observed reference chain is legitimate (such as when objects are kept in a cache in anticipation of future uses) or represents a leak. Third, the sheer size of the heap — large-scale server applications can easily contain tens of millions of objects — makes manual inspection of even a small subset of objects difficult or impossible.

Existing diagnosis tools are either online or offline. Online tools monitor either the state of the heap or accesses to objects in it or both. They analyze changes in the heap over time to detect leak candidates, which are "stale" objects that have not been accessed for some time. Online tools are not widely used in production environments, in part because their overhead can make them too expensive, but also because the need to debug memory leaks often occurs unexpectedly after an upgrade or change to a framework component, and often when developers believe their code has been sufficiently tested. Offline tools use heap snapshots, often obtained post-mortem when the system runs out of memory. These tools find leak candidates by analyzing the relationships, types, and sizes of objects and reference chains. Most existing heuristics, however, are based solely on the amount of memory an object retains and ignore structural information. Where structural information is taken into account, it often relies on prior knowledge of the application and its libraries.

Our work presents a graph mining approach to identifying leak candidate structures in heap dumps. Our approach is based on the observation that leaks often involve container data structures from which programmers fail to remove unneeded objects that were previously added. Consequently, the heap dump involves many subgraphs of similar structure containing the leaked objects and their descendants. By mining the dump, we can identify those recurring subgraphs and present the developer with statistics about their frequency

and location within the graph. Our key contributions can be summarized as follows:

1. Although analysis techniques are widely used in heap analysis [37, 38, 39], our work is the first to employ graph mining for detecting leak candidates. Specifically, we demonstrate that graph grammar mining used in an offline manner can detect both seeded and known memory leaks in real applications.

2. Compared to other offline analysis techniques, our approach does not require any *a priori* knowledge about which classes are containers, or about their internal structure. It captures containers even when these are embedded into application classes, such as ad-hoc lists or arrays.

3. Our approach can identify leaks even if the leaks' locations within the graph do not share a common ancestor node, or if the paths from that ancestor to the instances are difficult to find by the manual examination that is required in existing tools such as Eclipse Memory Analyzer (MAT).

4. Graph grammar mining can find recursive structures, giving a user insight into the data structures used in a program. For instance, linked lists and trees can be identified by their distinct signatures.

5. We use the dominator tree construct as a way to reduce the size of the heap dump graph before mining it. This allows for efficiency gains due to:

   - Space reduction in storing the input graph.
   - Runtime reduction due to the smaller amount of data to investigate.
   - Runtime reduction as a result of mining a tree. The unique structure of a tree enables a less complex mining process.
   - Simplified root path traversal.

   We claim that any directed graph dataset where nodes predominately have in-degree ≤ 1 can be easily preprocessed with the dominator tree computation without significantly affecting the results of the mining algorithm.

6. We expand upon the graph grammar inference algorithm presented in [27, 29], using sampling techniques to make it scalable for larger graphs.

7. Finally, the ability to combine subgraph frequency with location information makes our algorithm robust to the presence of object structures that occur naturally with high frequency without constituting a leak.

This work will be appearing in the 16th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, 2010 [36].

2.5.2 Biological Database Clustering

High-throughput technologies have greatly enhanced our ability to study biology from a top-down perspective. Instead of focusing on individual genes, molecules, or chemical reactions (now on referred to as biological entities or "biots" [2]) , we now view important biological processes as systems of interacting biots where it is more important to study the major behaviors and mechanisms of the entire system opposed to its individual components. This modern field of science is often called "systems biology" to reflect the study of biological phenomena as a system of cellular processes and machinery. Examples of high-throughput technologies include Next Generation Sequencing (NGS) methods [49], microarray gene expression analysis [32], yeast-two-hybrid screens for protein-protein interactions [17], chromatin immunoprecipitation on chip (ChIP-chip) for protein-DNA interactions [45], and along with many other modern technologies, RNAi screening [18]. RNAi screening is a specific high-throughput technology that is essential to our application, and it will be briefly described in Section 4.1.1.

The wave of development in high-throughput technologies has instigated an influx of computer science research in the field of bioinformatics. High-throughput experimentation produces massive quantities of raw biological data that must be analyzed for comprehension. Obtaining nontrivial knowledge from such large volumes of data requires extensive computational analysis. Bioinformatics is a growing field of science and is continuously driven by applications aiming to disseminate more information to fill the many gaps of uncertainty in biological systems and processes. Examples include inferring gene function from protein-protein interaction data, assembling full genomic sequences from massive quantities of short subsequences read from genome sequencing machines, protein folding prediction, extracting related sets of genes determined by gene expression and other measurements, and other applications that require computational analysis for efficient and robust examination.

The ongoing rapid advances in biotechnology and bioinformatics in particular have created a source for high-impact research. In this work, we will present a graph mining algorithm that we have created to analyze high-throughput biological data integrated from multiple sources into a publicly accessible relational database based on the *C. elegans* nematode. We design and implement an approximation algorithm for solving the optimal multipartite clique cover problem and use it as a tool for graph clustering and database summarization on the multipartite graph induced from this database. The main contributions of the work are as follows:

1. We show how a heterogeneous relational database can be used to construct a dense, multipartite graph.
2. We define the optimization problem of multipartite clique cover and present it as a novel approach to multipartite graph clustering.
3. We present an approximation algorithm to solve the optimal multipartite clique cover

problem.

4. Using the clusters extracted from our algorithm, we show that we can provide the user with a simplified view of the dataset consisting of disjoint sets of interrelated entities. We also exhibit example clusters and rationalize their correctness.

5. We analyze the performance of our algorithm in comparison to multiple variations of the method.

This work aims at providing scientists with a tool to enable the unsupervised exploration and reporting of groups of related entities that may be hidden within the depths of large databases. We show that the complex task of large-scale data comprehension can be greatly simplified with the use of clustering.

Chapter 3

Diagnosing Memory Leaks

The work presented in this chapter addresses a major problem affecting large software applications, especially persistent server applications where downtime can be costly. A memory leaks is a class of program bug that occurs when memory is allocated to the heap but does not become freed. Memory leaks can accumulate over time and eventually cause application crashes. We study memory leak detection and diagnosis in an offline fashion by introducing a graph grammar mining technique on heap dumps. In subsequent sections, we present more details about the causes and structure of memory leaks, how to generate a graph based on the heap dump of a Java application, and the techniques we use to mine the heap dump for memory leak sources. Finally, we present and discuss our algorithm's effectiveness on both synthetic and real-world Java applications.

3.1 Anatomy of a Leak

Although memory leaks can occur in all languages that use dynamically allocated memory, they are particularly prevalent in type-safe languages such as Java, which rely on garbage collection to reclaim memory. In these languages, dynamically allocated objects are freed only if the garbage collector can prove that no accesses to them are possible on any future execution path, which is true if and only if there is no path from a set of known roots to the object in the reachability graph. The reachability graph consists of nodes that represent allocated objects and edges that correspond to inter-object references stored in instance variables (fields). Roots, also known as garbage collection (GC) roots, are nodes whose liveness does not depend on the liveness of other heap objects, but on the execution semantics of the program. For instance, in Java, local and global static variables represent roots because the objects referred by them must remain reachable throughout the execution of a method or program, respectively. Hence, memory leaks form if objects remain reachable from a GC root even though the program will no longer access them. Such leaks also occur in programming

languages with explicit memory management, such as C++, and our work applies to them. We do not consider leaks arising from memory management errors in those languages (e.g., failing to deallocate unreachable objects).

```
public class HiddenLeak
{
  static HashMap legitimateMap;

  static class Legitimate
  {
    HashMap leakyMap = new HashMap();

    static class Leak
    {
      // This object is leaked
    }

    void leak()
    {
      // insert Leak instances into leakyMap
    }
  }

  public static void main(String []av)
  {
     // create N instances of Legitimate
     for (int i = 0; i < N; i++)
     {
       Legitimate legit = new Legitimate();
       legit.leak();
       legitimateMap.put(i, legit);
     }
  }
}
```

Figure 3.1: An example of a leak "hidden" underneath legitimate objects.

The Java program sketched in Figure 3.1 illustrates how leaks in a program manifest themselves in the reachability graph. In this example, a program maintains a number of objects of type `Legitimate`, which are stored in a hash map container. Each `Legitimate` instance, by itself, represents data the program needs to maintain and thus it needs to retain references to each instance. However, `Legitimate` also references a container `leakyMap` that accrues, over time, objects of type `Leak` that should not be stored permanently. Figure 3.2 shows the resulting heap structure. The hash maps exploit an open hashing scheme, which uses an array of buckets. Each bucket maintains a separate chain of entries corresponding

to keys for which hash collisions occurred.

Over time, the space taken up by the leaked objects will grow until all available heap space is exhausted. When this limit is reached, the Java virtual machine throws a runtime error (`OutOfMemoryError`). In many production environments, the JVM is run with a flag that saves a snapshot of the heap at this point, which is then fed to a heap analyzer tool.

Most existing tools compute and analyze the dominator tree of the heap object graph. If the object graph is a tree, as in this example, then it is identical to its dominator tree. In the example, the static (global) variable `legitimateMap` is the dominator tree root that keeps alive all leaked objects.

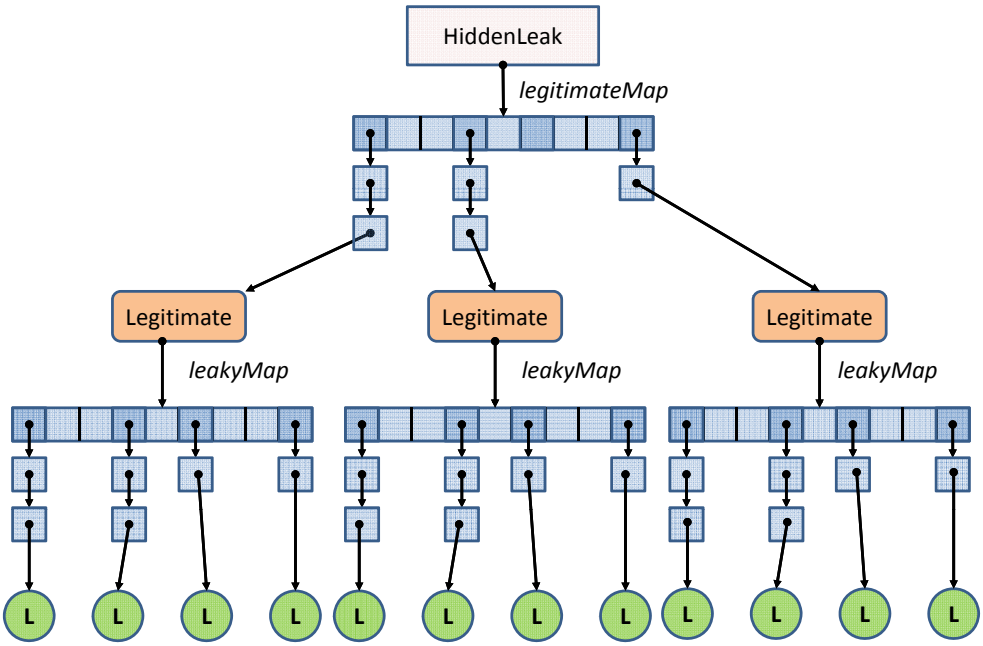


Figure 3.2: Heap graph after executing the program shown in Figure 3.1.

However, inspection of the dominator tree for this example with existing tools does not readily point to the leak. For instance, when examining a heap dump with the Eclipse Memory Analyzer tool (available at http://eclipse.org/mat), the tool pointed at 'legitimateMap' as a likely leak candidate, because it keeps alive a large fraction of the heap. None of its children stands out as a big consumer with respect to retained heap size. The leak is "hidden" under a blanket of legitimate objects. At this point, a developer would be required to "dig down," and individually examine paths through each Legitimate instance, which is cumbersome without global information about the structure of the subtrees emanating from these instances. Heap analyzers do support some global information, but it is usually limited to histogram statistics that shows how often objects of a certain type occur. This approach

often leads to limited insight because `String` objects and `char` arrays usually consume most memory in Java programs.

Consequently, there is a need to mine the graph to identify structures that are likely leak candidates, even if these structures are hidden beneath legitimate live objects. Moreover, developers require aggregate information that describe where in the object graph these leak candidates are located.

3.2 Heap Dump Processing

3.2.1 Generating a Heap Dump

We obtained heap dumps from Sun's Java VM (version 1.6) using either the jmap tool or via the `-XX:+HeapDumpOnOutOfMemoryError` option, which triggers an automatic heap dump when the JVM runs out of memory. We use the `com.sun.tools.hat.*` API to process the dump and extract the reachability graph. Each node in the graph corresponds to a Java object, which is labeled with its class. Each edge in the graph corresponds to a reference, labeled with the name of the field containing the reference. We label edges from arrays with `$array$`, ignoring the specific index at which a reference is stored. We remove all edges that correspond to weak and soft references, because weak and soft references to an object do not prevent that object from being reclaimed if the garbage collector runs low on memory.

3.2.2 Mining the Dominator Tree

Our approach is based on the observation that a leak would manifest as a heap dump containing many similar subgraphs. Rather than directly mine the heap dump, we compute the dominator tree of the heap dump and mine frequent graph grammars [27, 29] (see Algorithm 1) in the dominator tree. We describe the rationale behind this approach and algorithmic design decisions in this section.

As described in Section 2.3, the dominator tree of an input graph is a tree such that edges imply a reachability relation. In other words, a directed edge from a parent to a child in the dominator tree indicates that, in the original graph, the child node can only be reached via a path through the parent node. In order to compute a dominator tree, an entry node must be designated that will act as a root node of the input graph. The entry node is similar to a source node for network flow — it is where all paths through the graph begin. However, Java heap dump reachability graphs do not contain a single entry node, they contain a set of root nodes for garbage collection (see Section 3.1). We therefore introduce a pseudo-root node ρ to the input graph G, and connect it to the set of garbage collection roots GC (see Section 3.1) by adding edges $(\rho \rightarrow GC_i)$ where $GC_i \in GC$.

The dominator tree computation on the heap dump graph provides us with several benefits. First, it significantly reduces the number of edges in the graph to be mined. Second, since the resulting graph is a tree, we can apply optimizations in the mining algorithm specific to mining trees. Finally, recall that our goal is not just to find the frequent subgraph representing the leak but also to characterize the source of the leak. Once the dominator tree has been computed, we can search the paths from the entry node of the leaking subgraph up to the root of the dominator tree. This path is generally sufficient to identify the source of the leak. Without the dominator tree computation, tracing all paths to garbage collection roots in the graph is much more expensive and is full of noise.

In general, frequent subgraphs in the original heap dump need not necessarily be frequent in the dominator tree. To understand this, consider the cases in Figure 3.3 which shows example graphs that are frequent in both the dump and the dominator, frequent in the dump but not the dominator, as well as the other two combinations. In particular, the (frequent in dump, infrequent in dominator) combination occurs due to the existence of different routes of entry into a frequent subgraph S in graph G. This situates S into different subgraphs in D that may not be frequent individually. The reverse combination, i.e., (infrequent in dump, frequent in dominator), also happens, because frequent subgraphs in D may contain edges that summarize dissimilar paths in G.

Nevertheless, heap dumps have typical degree distribution properties that we can exploit. As we show later, a large majority of nodes in heap dumps have an in-degree of either zero or one. This implies that cases as shown in Figure 3.3 (top right) are much fewer in number than cases in Figure 3.3 (top left). This is a key distinction because we can guarantee that if a frequent subgraph S in G contains only nodes having in-degree ≤ 1, all instances of S will be completely conserved after the computation of the dominator tree D and will retain their frequencies. Although it is unlikely that a frequent subgraph in the heap dump will comprise *exclusively* of nodes with in-degree ≤ 1, our experience shows that it will be composed predominately of such nodes. These observations justify our design decision to compute the dominator tree as a preprocessing step before graph mining and to mine frequent structures in the dominator tree.

3.3 Extracting Leak Candidates

To mine patterns in the dominator tree, we explore the use of graph grammars [16, 48] instead of mere subgraphs. Graph grammars are necessary because leaking objects are often recursive in nature and we require the expressiveness of graph grammars. Furthermore, it is not necessarily the number of instances of a subgraph that is important in debugging the leak, but rather the percentage of the heap dump that is composed of instances of the subgraph. An algorithm that finds subgraphs could be misleading because a simple count of the number of instances could over-calculate the composition of the subgraph.

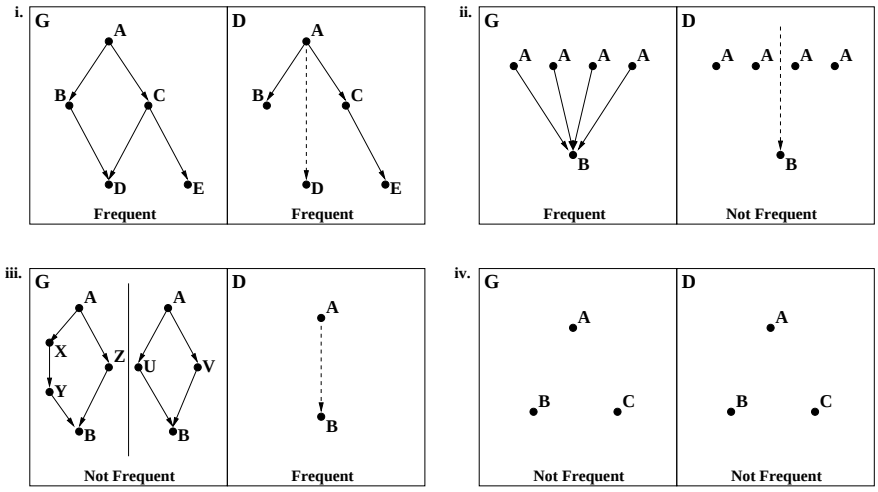


Figure 3.3: Examples of subgraphs that may be frequent or not frequent in either or both the graph G and its dominator tree D. (i-iv) show all 4 classes of subgraphs defined by the different combinations. Dashed edges represent a "dominates" edge produced only for D.

As described in Section 2.2.2, we build graph grammars in *a priori* fashion where we find a production rule capturing a significant portion of the input graph, replacing instances of the production by its non-terminal symbol. Candidate productions are evaluated for their ability to compress the graph. However, we found that the MDL heuristic does not scale well with large graphs and is impractical. We reasoned that the most significant contribution to the MDL score used for inference of graph grammars was the number of nodes ($|V|$) and edges ($|E|$) covered by the grammar's instances. We use the alternative size heuristic from [29]:

$$\frac{\text{size}(G)}{\text{size}(S) + \text{size}(G|S)}$$

where $\text{size}(G) = |V| + |E|$. The size heuristic is significantly much less computationally complex than the MDL heuristic but, in our experience, produces scores comparable to MDL.

Figure 3.4 shows two example tree input graphs G, a top-scoring graph grammar production S for each, and the result of compressing G using S (denoted by $G|S$). The first example shows how the graph grammar can describe the same information as a frequent subgraph. The second example demonstrates additional features of graph grammars. The graph grammar mining algorithm works iteratively, where in each iteration the top-ranked production is used to compress G. The next iteration repeats the process on the newly compressed version of G with non-terminal nodes. In practice, we run the algorithm for just a few iterations (≤ 3) because we focus on the top-ranked grammar productions when trying to identify a memory leak. For scalability reasons, we use sampling at several key states in the mining process.

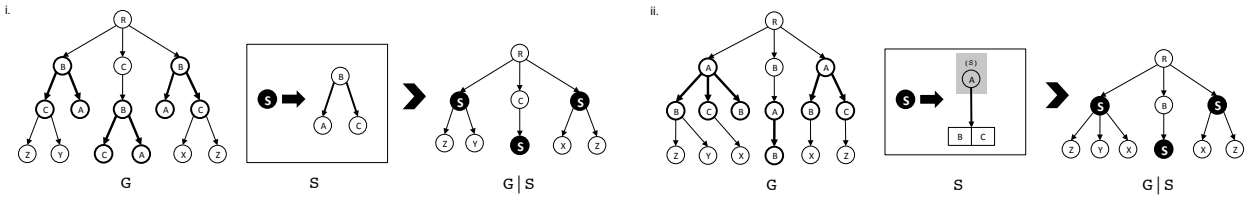


Figure 3.4: Two examples of an input graph G, a graph grammar S inferred from G, and the result of compressing G on S, denoted $G|S$. (i) A non-recursive grammar which does not contain any embedded non-terminal nodes or edges. (ii) Recursive grammar on node **A** containing an embedded non-terminal node that can match either nodes of type **B** or **C**. The grey box containing the "(S)" label represents a recursive connection instruction.

1. **Candidate generation:** When we expand an evaluated pattern with k edges to a candidate pattern with $k+1$ edges, we need not generate and explore all candidates because the algorithm uses a beam search to bound the number of patterns generated at each stage [29]. We take a random sample of $<1\%$ of the instances of the extending pattern to determine the best candidates to fill the beam. This sampling scheme provides efficiency gains to the procedure in line 8 of Algorithm 1 from Section 2.2.2. The full set of instances will then be explored only for the best candidates determined by the sample.

2. **Scoring candidates:** When a new candidate graph grammar production is generated, we must calculate the size heuristic of the candidate in the evaluation phase at line 11 of Algorithm 1 from Section 2.2.2. Since it will not have been checked completely for recursiveness yet (which will be explained below), we must ensure that we do not overestimate the size in the case that instances overlap. Instead of checking all instances of the candidate, we estimate the size heuristic by looking at a random sample of $\sim 5\%$ of all instances to get an idea for how much overlap occurs within the production's population. This estimate is used to approximate the actual score.

3. **Recursive Opportunities:** Similar to the sampling technique we use for scoring candidates, we sample the candidate production's instances to determine if and how it is recursive. This corresponds to the RecursifySubstructures function at line 9 of Algorithm 1 from Section 2.2.2. In this case, we use a sample of $<1\%$ of the instances. We can afford to use a smaller sample size than in the scoring function, because scoring requires a higher degree of accuracy.

Because the statistical significance of our sample would be largely dependent on the prevalence and recursive nature of a candidate graph grammar production, we cannot generalize our method to all cases and we choose our sample sizes empirically. However, our small sample sizes worked well in all of our experiments. We also note that once the top scored

graph grammar production is returned by the algorithm with sampling, we ensure that the recursion detection and compression of the input graph by the production are done exactly.

3.4 Evaluation

Our evaluation contains three parts. First, we check whether our algorithm finds seeded structures in a set of synthetically created dumps. Second, we examine its efficacy on heap dumps we obtained from Java developers. Third, we report its scalability and performance with respect to the sizes of the heap graphs considered.

3.4.1 Synthetic Examples

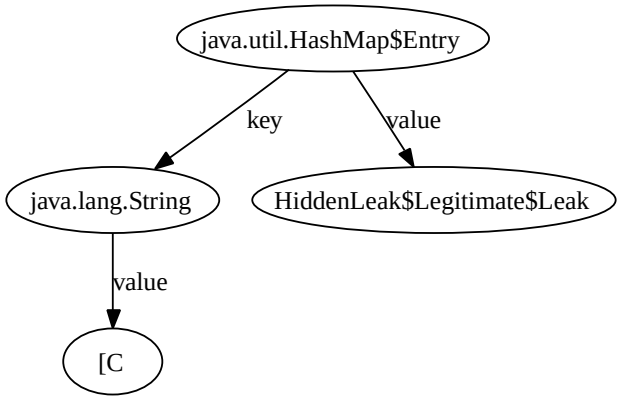


Figure 3.5: Most frequent grammar mined from HiddenLeak example in Figure 3.2.

We first present the results for the motivating example presented in Section 3.1. Figure 3.5 shows the most frequently occurring mined grammar, which represents a (key, value) pair anchored by an instance of type `HashMap.Entry`. This information directs an expert's attention immediately to a `HashMap` mapping keys of type `String` to values of type `HiddenLeak.Legitimate.Leak`. We found that 70% of the paths from the instances produced by this grammar to GC roots in the original graph exhibit the following structure, where `java.lang.Class` (top) is a root node of the dominator tree, `java.util.HashMap$-Entry` (bottom) corresponds to the top node in the best grammar displayed in Figure 3.5, and edge labels (in parentheses) stem from the explicit and implicit variable names used in the source code from Figure 3.1:

```
java.lang.Class
| (legitimateMap)
+-+-> java.util.HashMap
| (table)
+-+-> [Ljava.util.HashMap$Entry;
| ($array$)
+-+-> java.util.HashMap$Entry
| (value)
+-+-> HiddenLeak$Legitimate
| (leakyMap)
+-+-> java.util.HashMap
| (table)
+-+-> [Ljava.util.HashMap$Entry;
| ($array$)
+-+-> java.util.HashMap$Entry
```

The remaining paths contain an additional edge `Entry.next`, which represents the case in which a hash collision led to chaining. This information immediately describes the location of all instances of `Leak` objects in the graph, alerting the developer that a large number of these structures has accumulated underneath each `Legitimate` object. As discussed in Section 3.1, a size-based analysis of the dominator tree as done in Eclipse's Memory Analyzer would lead only to the bucket array of the `HashMap` object referred to by 'legitimateMap' and require manual inspection of the subtree emanating from it.

Since programmers often choose container types depending on specific space/time trade-offs related to an expected access pattern, we then investigated if the leaking structure would have been found if a different container type had been used. We replaced both uses of `HashMap` with class `TreeMap`, which uses a red-black tree implementation. Our algorithm correctly identified a grammar consisting of `TreeMap.Entry` objects that refer to a (key, value) pair, nearly identical to the grammar shown in Figure 3.5. In addition, the aggregated path was expressed by the recursive grammar shown in Figure 3.6, which covers over 99% of observed paths from a root to the grammar's instances. This path grammar identifies the leak as hidden in a tree of trees and provides a global picture that would be nearly impossible to obtain by visual inspection. The use of recursive productions enabled the algorithm to identify a classic container data structure (a binary tree) without any *a priori* knowledge.

The use of recursive grammars is also essential for other recursive data structures, such as linked lists. The following example demonstrates that our mining approach easily captures such data structures, even when they occur embedded in application classes (rather than in dedicated collection classes). Class `OOML` in Figure 3.7 embeds a link element `next` and an application-specific `payload` field. The main method contains an infinite loop that adds elements to a list held in a local variable 'root' until heap memory is exhausted.

Figure 3.8 shows the most frequent subgraph, which contains a recursive production

```
+-+-> java.lang.Class
| (legitimateMap)
+-+-> java.util.TreeMap
| (root)
{
   +-+-> java.util.TreeMap$Entry
   | ( right | left )
   +-+-> java.util.TreeMap$Entry
}*
| (value)
+-+-> leaks.TreeMapLeaks$Leak
| (leakyMap)
+-+-> java.util.TreeMap
| (root)
{
   +-+-> java.util.TreeMap$Entry
   | ( right | left )
   +-+-> java.util.TreeMap$Entry
}*
```

Figure 3.6: Resulting root path grammar if a TreeMap container is used for the example shown in Figure 3.2.

$OOML \rightarrow^{next} OOML$, representing a single linked list. The root path aggregation showed the location of its instances in the graph:

```
+-+-> Java_Local
| (root)
{
   +-+-> OOML
   | (next)
   +-+-> OOML
}*
```

3.4.2 Web Application Heapdumps

Apache Tomcat/J2EE

We obtained a series of heap dumps that resulted from recurring out-of-memory situations during the development of the LibX Edition Builder, a complex J2EE web application that makes heavy use of multiple frameworks [6], including the Apache Tomcat 5.5 servlet con-

```
public class OOML {
    OOML next;          // next element
    String payload;

    OOML(String payload, OOML next) {
        this.payload = payload;
        this.next = next;
    }

    // add nodes to list until out of memory
    public static void main(String []av) {
        OOML root = new OOML("root", null);
        for (int i = 0; ; i++)
            root = new OOML("content", root);
    }
}
```

Figure 3.7: A singly linked list embedded in an application class.

tainer. These heap dumps were generated over a period of several months. When the server ran out of memory during intense testing, developers would simply save a heap dump and restart the server, without immediate investigation of the cause.

We obtained a total of 20 heapdumps, varying in sizes from 33 to 47MB. In all of these dumps, the grammar shown in Figure 3.9 percolated to the top. This grammar represents instances of type `BeanInfoManager` that reference a `HashMap` through their `mPropertyByName` field. 80% of the root paths are expressed via the following grammar:

```
+-+-> org.apache.catalina.loader.StandardClassLoader
| (classes)
+-+-> java.util.Vector
| (elementData)
+-+-> [Ljava.lang.Object;
| ($array$)
+-+-> java.lang.Class
| (mBeanInfoManagerByClass)
+-+-> java.util.HashMap
| (table)
+-+-> [Ljava.util.HashMap$Entry;
| ($array$)
+-+-> java.util.HashMap$Entry
| (value)
+-+-> org.apache.commons.el.BeanInfoManager
```

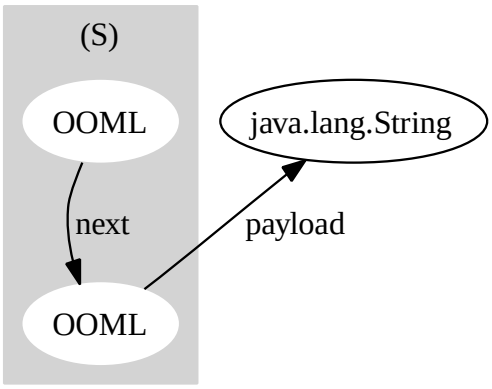


Figure 3.8: Most frequent grammar in synthetic linked list example.

This grammar shows that the majority of these objects are kept alive via a field named `mBeanInfoManagerByClass`. Since the field is associated with a node of type `Class`, it represents a static field. Examination of an actual path instance reveals that this static field belongs to class `org.apache.commons.el.BeanInfoManager`.

Similar to the HiddenLeak example, MAT reported the (legitimate!) `HashMap.Entry` array stored in the `mBeanInfoManagerByClass` class as an accumulation point, but could not provide insights into the structure of the objects kept in this table without tedious manual inspection. We eventually found that this leak had already been reported by another developer against the Tomcat Apache server (Bug 38048: Classloader leak caused by EL evaluation) [1]. Interestingly, the original bug report had received little attention, likely because the bug reporter included only a single trace to a leaked object reachable from the `BeanInfoManager` class.

This leak was subsequently fixed in a 6.x release of Tomcat. After updating the server, we periodically took heap dumps via the jmat tool. We subjected these heap dumps, which contain no known leaks, to our analysis. Unsurprisingly, the subgraph anchored by `HashMap.Entry` rose to the top, reflecting the ubiquitous use of hash maps. However, path aggregation showed these hash maps were located in very different regions of the object graph, thus making it less likely for them to be leaks.

MVEL

In a separate project, a rule-based system was developed for the application of software engineering patterns to enhance code [51]. During the development of this project, out of memory situations occurred when certain input was fed to the rule engine, which was written using the Drools rule engine framework. In this system, rules can contain expressions written in the MVEL scripting language (mvel.codehaus.org).

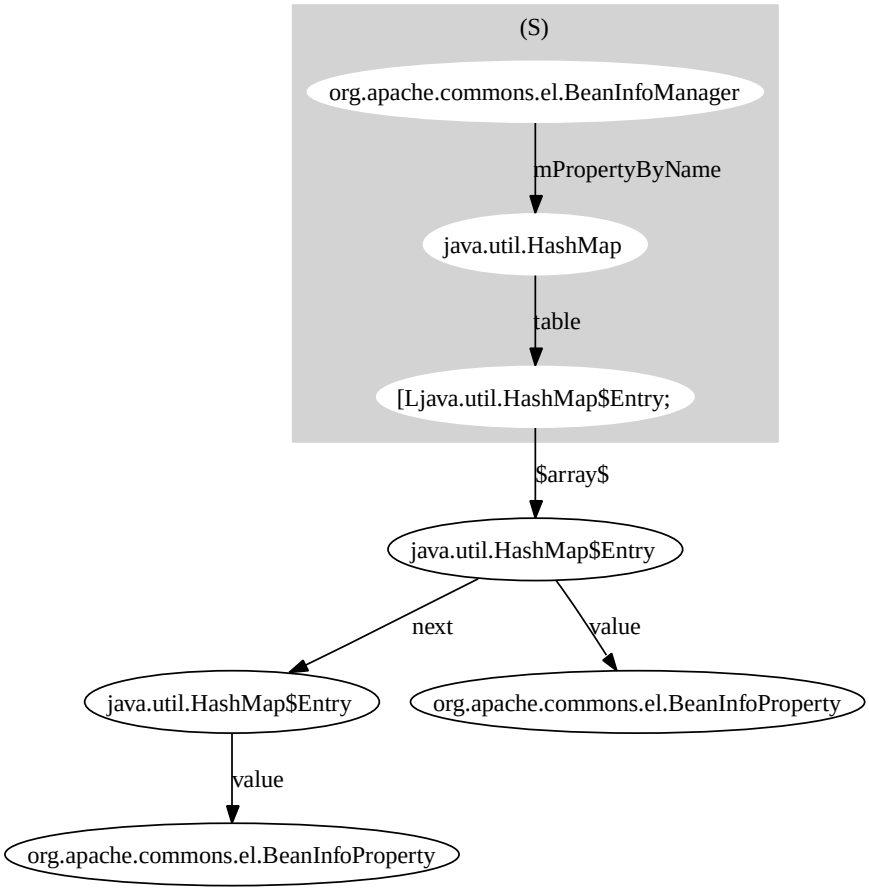


Figure 3.9: Most frequent grammar in Tomcat 5.5 heap dump.

Mining the resulting heap dump showed the grammar in Figure 3.10, which contains a recursive production $RegExMatch \rightarrow^{nextASTNode} RegExMatch$. This mined grammar mirrors the synthetic linked list discussed in Section 3.4.1. All RegExMatch objects are contained in a single list held in a local variable:

```
+-+-> Java_Local
| (??)
+-+-> org.mvel.ASTLinkedList
| (firstASTNode)
{
   +-+-> org.mvel.ast.RegExMatch
   | (nextASTNode)
   +-+-> org.mvel.ast.RegExMatch
}*
```

This path indicates that the cause of the memory exhaustion was the unbounded growth of a singly-linked list of `RegExMatch` objects, likely due to a bug in the MVEL parser. Although this information does not directly lead to the underlying bug, it rules out a number of other scenarios, such as memory exhaustion due to a large object kept alive by a long list of `RegExMatch` objects.

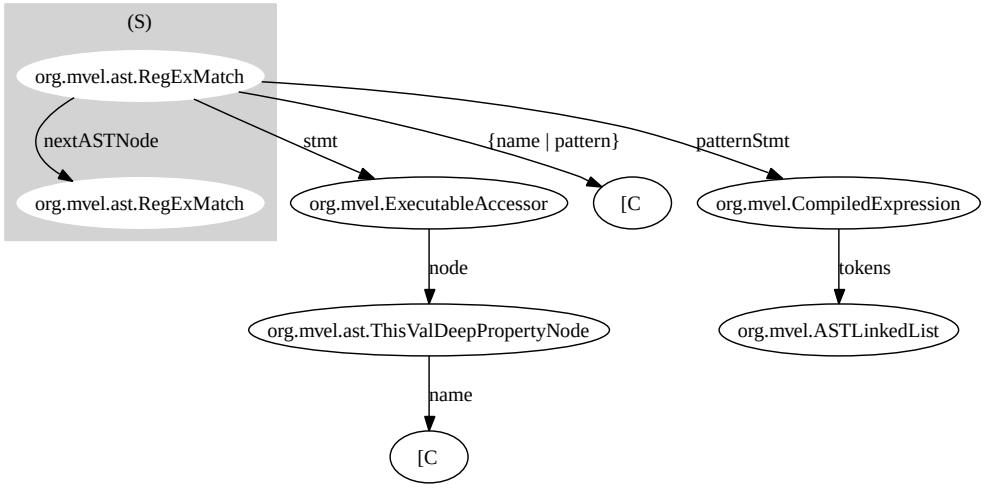


Figure 3.10: Most frequent grammar in MVEL heap dump.

3.4.3 Scalability and Other Quantitative Aspects

In this section, we study some quantitative aspects of our graph mining approach to illustrate its effectiveness at mining heap dumps. First, we study the indegree distribution of nodes from 24 real (i.e., not synthetic) heap dumps. Assessing the percentage of nodes that have in-degree ≤ 1 across these dumps, we obtain statistics of a minimum of 84%, an average of 89%, and a maximum of 99.8% (from the MVEL dumps). This suggests that assessing frequent subgraphs in the dominator tree should not cause significant loss of information as compared to the original heap dump. At the same time, Table 3.1 illustrates the reduction gained in the number of edges and the overall size of the graph by choosing to focus on the dominator tree.

Figure 3.11 illustrates the time taken for diagnosing leaks as a function of the size of the dominator. This time does not include loading and pre-processing (removal of weak and soft references) and computation of the dominator. It does include the time to mine the top (best) graph grammar, for graph reduction, and for summarizing root paths. The lower cluster of points is drawn from the Tomcat/J2EE dumps. These are processed faster because they do not involve recursive constructs and there are fewer instances of the mined grammar in the dump. We see that these are processed in about 2–3 minutes. Conversely, the MVEL dumps

Table 3.1: Summary statistics of some of the heap dumps analyzed in this work. Each heap dump is named by the type of leak it contains and the date it was created. Dumps labeled as 'maintenance' were taken before the production server was shut down for maintenance; they appear to be healthy and leak-free. Grammar size is reported as an average across three test runs that may differ due to the non-determinism of the algorithm when using random sampling.

Name	**Memory (MB)**	**# nodes**	**# edges (G)**	**# edges (D)**	**% edge reduction**	**% size reduction**	**Grammar size (avg)**
tomcat.jul03	39	713076	1243152	536628	57%	36%	10.3
tomcat.jul05	41	732089	1288883	555829	57%	36%	7.0
tomcat.jun02	33	603941	1035000	440201	57%	36%	11.7
tomcat.jun04	33	588559	974322	427347	56%	35%	9.7
tomcat.may02	41	745434	1442146	561377	61%	40%	31.0
tomcat.may15	40	788482	1545443	609702	61%	40%	31.0
tomcat.sep20	40	734027	1295724	561956	57%	36%	10.3
tomcat.oct20	36	655251	1133970	487573	57%	36%	14.3
tomcat.nov05	43	742729	1336582	559642	58%	37%	24.3
tomcat.oct24	41	707913	1218834	533087	56%	36%	11.3
tomcat.nov03	42	715032	1202226	522893	57%	35%	23.7
tomcat.nov06	42	710730	1250427	535371	57%	36%	25.7
tomcat.oct28	42	717464	1260223	540283	57%	36%	9.0
tomcat.oct06	38	700479	1212796	530758	56%	36%	10.3
tomcat.oct03	47	854365	1584715	674692	57%	37%	11.0
tomcat.oct14	40	722417	1278437	550516	57%	36%	10.3
maintenance.feb06	34	321786	464017	260367	44%	26%	15.0
maintenance.nov08	35	519435	635638	348284	45%	25%	11.0
maintenance.nov17	35	547759	689958	374595	46%	25%	11.0
maintenance.nov09	34	493408	577236	322739	44%	24%	14.7
mvel.feb12	69	1704284	3832262	1698297	56%	39%	15.0
mvel.feb13	69	1704286	3832264	1698296	56%	39%	13.0
mvel.feb14	69	1704810	3832897	1698405	56%	39%	13.0
mvel.feb19	69	1707258	3837542	1701496	56%	39%	15.0

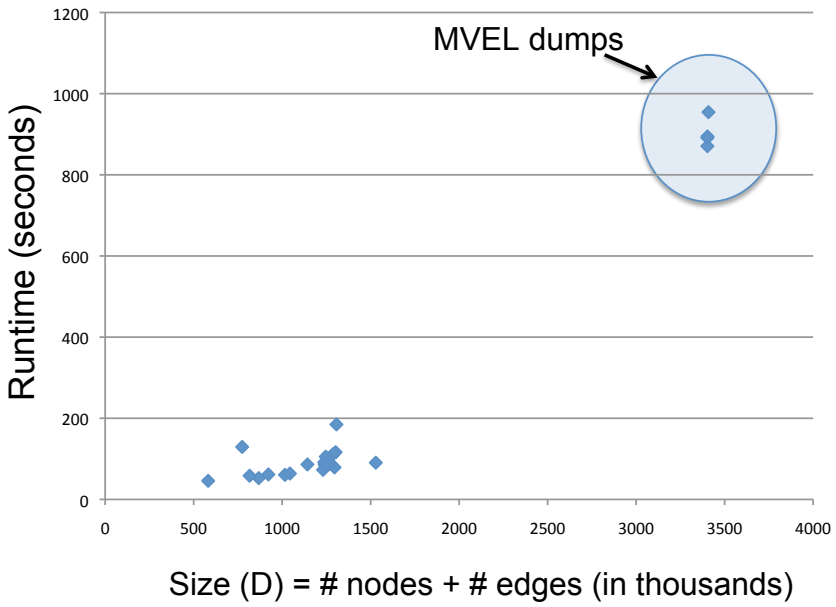


Figure 3.11: Runtime statistics on real heap dumps.

involve significant recursion and several hundreds of thousands of instances. Furthermore, the path summarization for the MVEL constructs require greater work since we must traverse up the linked list to observe the root path for even a single instance. With MAT, one of the smallest heap dumps (tomcat.jun04) took 11 seconds to process the dominator tree and an additional 6 seconds to produce the leak suspects report, totaling 17 seconds. On one of the larger mvel dumps, the dominator tree was computed in 17 seconds, and the leak suspects report in 5 seconds, totaling 22 seconds. Thus MAT provides a quicker analysis, but the reports contain much less detail about the leak compared to our method.

Finally, we compared the runtime of our algorithm when used on the dominator tree versus the original heap dump graph. We chose one particular dump, tomcat.sep20 from Table 3.1, to make the comparison. We used this dump because the Tomcat leak graph grammar does not display significant recursion and this dump is one of many good representatives of the leak class. In order to compare the runtimes, we excluded the path summarization step that was included in the runtime plot in Figure 3.11, because this step would not be applicable in the heap dump graph. We note that the preprocessing step of removing weak and soft references differs between graphs as well (in fact it is less complex in the dominator tree), but the resulting graphs are comparable. Further, we found that the most frequent graph grammar production in the dominator tree is a subgraph of the most frequent graph grammar production in the full heap dump graph, thus requiring more iterations of candidate generation and therefore more runtime for discovery. To enable a fair runtime comparison, we considered the time required to generate the most frequent single-edge graph grammar production from the dominator tree versus from the full graph. We

found that the dominator tree accomplished this task in 21 seconds versus 38 seconds in the full graph. We also compared the complete runtime for the graph grammar in the full heap dump graph for tomcat.sep20 versus the runtime on a dominator tree from a similar heap dump, tomcat.nov05, because the discovered graph grammar from the dominator tree in tomcat.nov05 was closer in size to that found in the full heap dump graph for tomcat.sep20 and composed of the same leak but was actually larger and more frequent. We found that in the dominator tree the runtime was 130 seconds versus 426 seconds in the full heap dump graph. Our results show that we obtain $\sim$46% runtime reduction for identifying small graph grammars and $\sim$69% runtime reduction for large graph grammars when the percentage of size reduction is only 36%. This suggests that mining the dominator tree is not only quicker due to size reduction, but also because its tree structure contains less noise and redundancy, thereby simplifying the mining process.

3.5 Related Work

Our research combines ideas from software engineering and data mining. We discuss related work in each of these areas.

3.5.1 Memory Leak Detection Tools

One of the first systems to debug leaks exploited visualization, allowing a user to interactively focus on suspected problem areas in the heap [43]. Most recent existing leak detection tools use temporal information, including object age and staleness, that is obtained by monitoring a program as it runs. For instance, IBM's Leakbot [37, 38, 39] acquires snapshots at multiple times during the execution of a program, applies heuristics to identify leak candidates, and monitors how they evolve over time.

Minimizing both the space and runtime overhead of dynamic analyses has been the subject of intense study. Space overhead is incurred because object allocation sites and last access times must be recorded; runtime overhead exists because this information must be continuously updated. Bell and Sleigh [8] use a novel encoding to minimize space overhead for allocation sites. Statistical profiling approaches were developed in [24] to minimize runtime costs. Cork [28] combines low-overhead statistic profiling with type-slicing. Some profilers, notably the NetBeans profiler, use information already kept by generational collectors to determine object age. Lastly, a hardware support strategy was proposed in [53] for monitoring memory access events.

By contrast, our approach explores mining information from a single heap dump, which is often the only source of information available when out-of-memory errors occur unexpectedly, as is a common case in production environments in which dynamic tools are rarely deployed. Our work is complementary to dynamic approaches. Mined structural information

is likely to enhance information these tools can provide, especially in the common scenario in which software engineers diagnose suspected leaks in codes with which they are not familiar. Moreover, the ability to identify data structures could be exploited to automatically infer which operations are add/delete operations on containers, which could benefit approaches that rely on monitoring the membership of object containers to identify leaks [56].

In the context of languages with explicit memory management, several static analyses have been developed that identify where a programmer failed to deallocate memory [12, 25, 41, 55]. Similarly, trace-based tools such as Purify [23] or Valgrind [40] can identify unreachable objects in such environments. By comparison, the garbage collected languages at which our analysis aims do not employ explicit deallocation; we aim to identify reachable objects that are unlikely to be accessed in the future. Lastly, rather than eliminating the source of leaks, some systems implement mitigation strategies such as swapping objects to disk [7].

3.5.2 Data Mining for Software Engineering

Data from programming projects (code, bug reports, documentation, runtime snapshots, heap dumps) are now so plentiful that data mining approaches have been investigated toward software engineering goals (see [54] for a survey). Graph data, in particular, resurfaces in many guises, such as call graphs, dependencies across subprojects, and heap dumps. Graph mining techniques have been used minimally for program diagnosis. For instance, program behavior graphs have been mined for frequent closed subgraphs that become features in a classifier to predict the existence of "noncrashing" bugs [34]. Behavior graphs were also mined with the LEAP algorithm [57] in [10] to identify discriminative subgraphs signifying bug signatures. However, to the best of our knowledge, nobody has investigated the role of mining heap dumps for detecting memory leaks or used a graph grammar mining tool.

Chapter 4

Clustering Biological Data

In this chapter, we present our work on clustering of heterogeneous data from *Caenorhabditis elegans* stored in the Computational Models for Gene Silencing (CMGS) database [2, 42]. We first provide background information about the database to contextualize the application, then we show how it is converted into a multipartite graph, and finally we present our graph mining algorithm and the results of running it on the CMGS database.

4.1 Computational Models for Gene Silencing Database

4.1.1 Background

C. elegans is a microscopic worm, a multicellular eukaryotic model organism that has been studied extensively by biologists. It is particularly significant to study multicellular eukaryotic organisms, because humans are in this classification. *C. elegans* is a useful model organism, because it is a very easy host to perform RNA interference (RNAi) experiments on. RNAi is a process by which short single or double stranded RNA complementary to a target gene is introduced into the organism. The complementary RNA binds to the transcripts of the target gene and degrades them so that they cannot be translated into protein and expressed. This process is commonly called "gene silencing" or "gene knockdown" because it silences the expression of a specific gene without modifying the host DNA. *C. elegans* is an especially good host for performing RNAi experiments, because it can be fed genetically altered bacteria containing the complementary RNA, which is an extremely convenient and simple method for inducing RNAi. A large percentage of the *C. elegans* genome has been studied through gene knockdowns, and biologists now have a better understanding of the function of many genes as well as their essentiality for viability of the organism [3, 9].

Following the analysis of the *C. elegans* genome through RNAi, a large amount of data pertaining to gene-phenotype relationships was collected and deposited into the WormBase

online database [3, 21]. These data, along with protein-protein interactions, protein orthologs, gene regulation, gene ontology classifications, and pathways, were integrated into the CMGS database [2, 42]. These include the RNAi data we have described as well as other key relationships found using other state-of-the-art biological methods. These data integration studies are a common initiative in modern biology, and they provide a powerful repository for exploration. The generation and integration steps alone for building these repositories are major works and include minimal if any data analysis. We aim to perform data analysis on the CMGS database in order to yield insight into the many relationships hiding in the expanse of data integrated from the multiple sources.

4.1.2 Graph Structure and Generation

Using the CMGS database [2], as described in Section 2.4, we extract a large portion of the data and build a multipartite graph $\mathcal{MG} = (V, E)$. Figure 4.1 displays an entity-relationship diagram of the information we use from CMGS to build $\mathcal{MG}$. The contents of the edges for each relationship are described below:

Genes ↔ Phenotypes:
Gene silencing experiments (RNAi) are used to determine functional information about genes. If silencing a gene causes a specific phenotype, this is recorded as an edge between the gene and phenotype. If no change occurs, the gene has an edge to a "wild-type" phenotype, meaning the worm mutant appeared normal.

Genes ↔ Proteins:
Generally, a gene is transcribed into a specific RNA transcript that becomes a specific type of protein. It is not uncommon, however, for multiple genes to encode different parts of a protein complex. It is also possible for a gene to encode multiple protein types. This is even more common in humans where alternative splicing can occur. These edges in our graph represent the relationship of genes encoding proteins. In addition, we wanted to incorporate edges for gene regulation and protein-protein interactions, but these relationships would violate the multipartite graph property. Thus, for every gene regulation, we added edges between genes and the proteins that correspond to the genes that they regulate or are regulated by. Similarly for protein-protein interactions, we added edges between the genes and proteins that correspond to each interaction. These implicit relationships conserve the gene regulation and protein-protein interaction information without violating the multipartite graph property.

Genes ↔ GO categories:
The Gene Ontology (GO) is a directed acyclic graph where nodes (GO categories) represent classifications of either cellular components, molecular functions, or biological processes. GO categories are used for annotating genes and are hierarchical. These edges in our graph represent a gene being annotated with a particular GO category.

Phenotypes ↔ GO categories:
Phenotypes and GO categories are both terminology that represent specific behaviors or biological characteristics of an organism. Accordingly, there may be relationships among GO categories and phenotypes, and these are encoded as edges in our graph.

Genes ↔ Pathways:
A pathway is a set of cascading chemical reactions that occur between molecular species in a cell to produce some function. For example, a metabolic pathway might take some molecule as input that catalyzes a set of chemical reactions that in turn degrade the molecule. Many of the molecular species in pathways are encoded by genes in the host DNA. Pathways are important to understand, because they help to describe dependencies between genes and they contextualize the importance of certain genes with respect to biological function. We include edges in our graph between a gene and a pathway if the gene is a participant in the pathway.

Genes ↔ Species:
A common method for determining function and other characteristics of a gene is to identify some other well-studied gene in a different species that is very similar in terms of either sequence, structure, or chemical composition. This similarity relationship is known as orthology and differs from paralogy where multiple genes in the same species are similar. If an ortholog for a gene is discovered, then the function and other biological information from the ortholog can be used to infer characteristics about the query gene. We add edges between genes and species if the gene has an ortholog in that species. This data originates from identification of orthologous proteins, but we present the orthology relationship between the genes that each protein encodes to maintain the gene-centric structure of the multipartite graph.

Table 4.1 shows the number of nodes and edges of each type contained in $\mathcal{MG}$. Note that we prevent multiple edges from occurring. In Table 4.1(b), we also include a density score to indicate what percentage of all the possible edges actually occur. Formally, for a given bipartite graph between $V_i, V_j \in \mathcal{MG}$, the density $\mathcal{D}_{ij} = \frac{|E_{ij}|}{|V_i \times V_j|}$ such that $|E_{ij}|$ is the number of edges between two arbitrary partition blocks i, j and $|V_i \times V_j|$ is the number of possible edges between them. $|V_i \times V_j|$ is equal to $|V_i| \times |V_j|$. Thus, $\mathcal{D}_{ij} = 1$ indicates that $E_{ij} = V_i \times V_j$, and $\mathcal{D}_{ij} = 0$ indicates that $E_{ij} = \emptyset$.

The $\mathcal{D}_{ij}$ values in Table 4.1(b) indicate that $\mathcal{MG}$ is relatively sparse. This should be expected, however, because biots should only have relationships with a minimal number of other biots or else the relationships would be nonspecific. It is not surprising that the genes-species relationship is the most dense as many genes will have orthologs within the seven species included. It is also not surprising that the density of relationships between phenotypes and GO categories is very sparse because, in general, phenotypes and GO categories are not synonymous. Further, the genes-proteins relationship is expected to be very sparse, because there are many genes and many proteins, but the encoding of proteins, gene regulations,

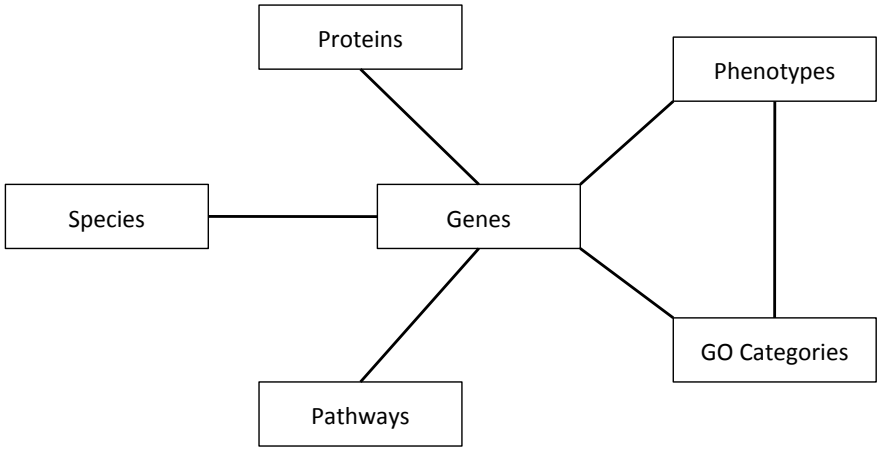


Figure 4.1: The adjacency information of the multipartite graph we pull from the CMGS database [2]. All are considered to be many-to-many relationships, some of which (genes-species) are more dense than others (genes-proteins) as shown in Table 4.1(b).

and protein-protein interactions are uncommon relative to all of those that are possible (i.e., in general, it should be close to a one-to-one relationship). However, this measure does not indicate how common bicliques are within $\mathcal{MG}$. We have found that they are in fact common but will contain a small subset of the nodes within each block.

4.1.3 Clusters of Multipartite Cliques

In Section 2.4.2, we introduce the problem of finding the optimal multipartite clique cover (MCC) and note that it is NP-hard. In Section 4.2, we present an approximate algorithm for MCC as well as our method to compute large multipartite cliques efficiently, but first we introduce the significance of this problem on our graph from the CMGS database.

In the context of our graph, a multipartite clique represents sets of biots that have a strong relationship among them. For instance, if we find a multipartite clique that contains nodes in the partition blocks of genes, proteins, pathways, and phenotypes, this might indicate that there is a set of genes that are all involved in the same pathway(s), cause the same phenotype(s) when knocked down, and encode or interact with the same proteins. This might be very interesting to a biologist, especially if the set of relationships is previously unrecognized. This could suggest transitive relationships, such as the perturbation of any number of genes in a certain pathway creates inviability in the organism. We would also gain the context that not only are the genes essential to the organism, but the specific protein complexes that they encode may be elements of a fundamental pathway that cannot be perturbed in order for the organism to survive.

We anticipate that the multipartite cliques that we find, which we consider clusters in the

Table 4.1: Distribution of nodes (4.1(a)) and edges (4.1(b)) in our multipartite graph, $\mathcal{MG}$.

(a) Node distribution of $\mathcal{MG}$.

Node Type	**# of Nodes ($\lvert V_i \rvert$)**
Genes	21393
Phenotypes	565
GO Categories	2119
Proteins	23037
Species	7
Pathways	110
Total	**47231**

(b) Edge distribution of $\mathcal{MG}$.

Edge Type	**# of Edges ($\lvert E_{ij} \rvert$)**	$\mathcal{D}_{ij}$
Genes $\leftrightarrow$ Species	95662	0.639
Genes $\leftrightarrow$ Phenotypes	57949	0.00479
Genes $\leftrightarrow$ Pathways	1778	0.000756
Genes $\leftrightarrow$ GO Categories	31473	0.000694
Genes $\leftrightarrow$ Proteins	38560	0.0000782
Phenotypes $\leftrightarrow$ GO Categories	88	0.0000735
Total	**225510**	**0.000407**

input graph, will likely contain many interesting, biologically meaningful relationships that span data collected from unrelated sources. We would like to find the optimal multipartite clique cover because this should correspond to the smallest set of clusters and would therefore contain, on average, the largest clusters. However, because we do not allow overlap between edges in clusters, we cannot guarantee that every cluster is maximal with respect to the input graph. Figure 4.2 shows this dilemma. In this case, there are two large multipartite clusters; one spans only between partition blocks P_1 and P_2 (green and black edges), while the other spans between all blocks P_1, P_2 and P_3 (blue and black edges). The first cluster contains 25 edges and the second cluster contains 21 edges. Theoretically, the first cluster is larger and would be preferred over the second cluster. As a result, the cluster containing only blue edges would be separated from the black edges, which may contain useful information. Similarly, if we chose the second cluster of 21 edges first, only the green edges are then clustered together. Thus, we cannot guarantee that we report all possible clusters, but we aim to find the best set of disjoint clusters.

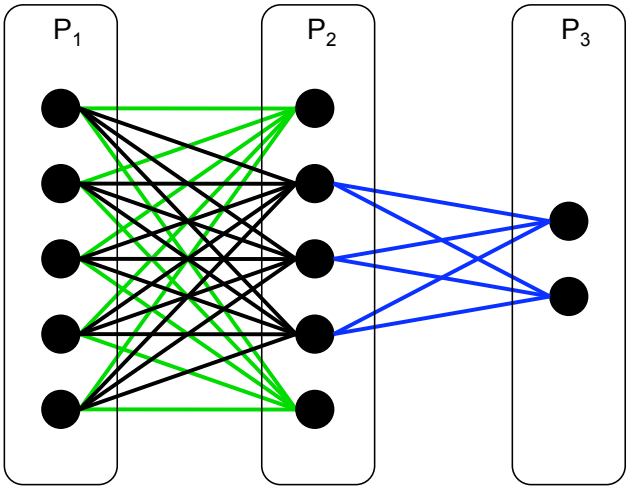


Figure 4.2: Shows two overlapping multipartite cliques, where black edges occur at the overlap of the green and blue cliques.

4.2 Approximate Algorithm for Multipartite Clique Cover

4.2.1 Method

In this section, we will present our multipartite graph clustering algorithm aimed at finding the optimal multipartite clique cover of the input graph. Our algorithm is theoretically based on a greedy optimal set cover approximation algorithm where sets containing the most uncovered items are chosen one at a time until all items are covered. Thus the algorithm is greedy because it makes the best local decision in hopes that it will be the best global decision. Our sets correspond to multipartite cliques where the items are edges in the graph. The main complexity in our algorithm is therefore finding the largest multipartite clique in a given iteration. In Section 2.4.1, we have noted that finding an edge-weighted maximal multipartite clique is an NP-complete problem, which motivates us to use heuristics to find large multipartite cliques that are not necessarily optimal.

To find a large multipartite clique, we pick a node in the graph with the largest degree (i.e., that has the largest number of nodes in its immediate neighborhood) and refer to it as the seed node s with P_s being its containing partition block. From s, we aim to add large bicliques that span to new blocks in an *ad hoc* fashion, until we have built a sufficiently large multipartite clique. This particular heuristic is based on the fact that finding the maximal edge biclique is known to be an NP-complete problem [44]. Therefore we start by building a large but not necessarily maximal biclique of size $1 \times N$, where N is the number of nodes in some block that neighbors P_s, and we choose the block that gives us the maximum N. We call the set of nodes N the *extension set* and refer to it as $\mathcal{E}_P(s)$ (see Table 4.2) because it consists of the set of nodes extended directly from s. We then try to span this biclique to new

blocks if it increases the edge coverage (i.e. number of edges) of the multipartite clique. Our motivation for starting with a biclique of size $1 \times N$ is that in random bipartite graphs, the maximal $1 \times N$ biclique tends to be significantly larger than the maximal balanced biclique (same number of nodes from each block) as described in [14]. Formally, this tells us that for the maximal $1 \times N_1$ biclique and maximal $N_2 \times N_2$ balanced biclique of a random bipartite graph, in general, $N_1 \gg (N_2)^2$. This suggests that we can build larger bicliques stemming from a single, high-degree seed node to its extension set rather than a more balanced biclique (that would also be more complicated to identify).

Formally, we present three algorithms to first find a suitably large multipartite clique given the input graph (Algorithms 3, 4) and, second, to perform this step iteratively to approximate the optimal multipartite clique cover (Algorithm 2). We define terminology in Table 4.2, and present the algorithms below.

Table 4.2: Algorithm terminology.

$\mathcal{MG}$	A multipartite graph.
P_a	The partitioned *block* of nodes in $\mathcal{MG}$ containing node a.
$N(u)$	The *neighborhood* of a node u is the set of nodes such that for all $v \in N(u), (u, v) \in E$ (the edge set). The neighborhood of a set of nodes U is defined as $N(U) = \bigcup_{u \in U} N(u)$.
$N_P(u)$	The *P-neighborhood* of a node u such that $N_P(u) \subseteq N(u)$ and contains only the neighboring nodes in partition block P (i.e., $N_P(u) = N(u) \cap V_P$). The P-neighborhood of a set of nodes U is defined as $N_P(U) = \bigcup_{u \in U} N_P(u)$.
$\mathcal{E}_P(u)$	The *extension set* of a node u such that $\mathcal{E}_P(u) \subseteq N_P(u)$ about a block P. The extension set of a set of nodes U is defined as: for all $v \in \mathcal{E}_P(U)$ and for all $u \in U$, $(u, v) \in E$ (the edge set of $\mathcal{MG}$). Thus $U \times \mathcal{E}_P(U)$ must form a biclique.
$\omega(\mathcal{C})$	The *coverage* or *weight* of a multipartite clique $\mathcal{C}$ is the number of edges contained in the clique. For a biclique B_{ij} having $\lvert b_i \rvert$ nodes in block P_i and $\lvert b_j \rvert$ nodes in block P_j, then $\omega(B_{ij}) = \lvert b_i \rvert \times \lvert b_j \rvert$. A multipartite clique $\mathcal{C}$ is composed of a set of bicliques $\mathbb{B}$, thus $\omega(\mathcal{C}) = \sum_{B \in \mathbb{B}} \omega(B)$.

We do not formally define the function GetSeedNode called on line 2 of Algorithm 2. The behavior of this function, as described previously, is to pick a node with the largest neighborhood. Thus we select:

$$s = \underset{n \in V_{\mathcal{MG}}}{\operatorname{argmax}} |N(n)|.$$

Algorithm 2 MULTIPARTITECLIQUECOVER($\mathcal{MG}$):

Input: A multipartite graph $\mathcal{MG}$.
Output: A set of multipartite cliques $\mathbb{C}$ that cover the majority of edges in $\mathcal{MG}$.

$\mathcal{MG}^{'} \leftarrow \mathcal{MG}$, $\mathbb{C} \leftarrow \emptyset$
$s \leftarrow$ GETSEEDNODE($\mathcal{MG}^{'}$)
while there exists a P s.t. $N_P(s) > 1$ **do**
  $\mathcal{C} \leftarrow$ EXTRACTMULTIPARTITECLIQUE($\mathcal{MG}^{'}, s$)
  $\mathcal{MG}^{'} \leftarrow \mathcal{MG}^{'} \setminus \mathcal{C}$
  $\mathbb{C} \leftarrow \mathbb{C} \cup \mathcal{C}$
  $s \leftarrow$ GETSEEDNODE($\mathcal{MG}^{'}$)
end while
return $\mathbb{C}$

Algorithm 3 EXTRACTMULTIPARTITECLIQUE($\mathcal{MG}, s$):

Input: A multipartite graph $\mathcal{MG}$, and a seed node $s \in \mathcal{MG}$.
Output: A multipartite clique $\mathcal{C}$ having at least 2 partition blocks containing s.

$\mathcal{E}_P(s) \leftarrow \underset{N_P(s)}{\operatorname{argmax}} |N_P(s)|$
$\mathcal{C} \leftarrow s \cup \mathcal{E}_P(s)$
$\hat{\mathcal{C}} \leftarrow$ BESTEXTENSION($\mathcal{MG}, \mathcal{C}, s, P_s$)
repeat
  $\mathcal{C} \leftarrow \hat{\mathcal{C}}$
  $\hat{\mathcal{C}} \leftarrow$ BESTEXTENSION($\mathcal{MG}, \mathcal{C}, \emptyset, \emptyset$)
until $\hat{\mathcal{C}} = \mathcal{C}$
return $\mathcal{C}$

Algorithm 4 BESTEXTENSION($\mathcal{MG}, \mathcal{C}, s, P_s$):

Input: A multipartite graph $\mathcal{MG}$, an initial clique to expand upon $\mathcal{C}$, and if being called on the initial biclique, s is the seed node and P_s is the block containing s. Otherwise $s = \emptyset, P_s = \emptyset$.

Output: A multipartite clique $\hat{\mathcal{C}}$ such that $\hat{\mathcal{C}} = \mathcal{C}$ if no good extension exists, otherwise $\omega(\hat{\mathcal{C}}) > \omega(\mathcal{C})$.

$\hat{\mathcal{C}} \leftarrow \mathcal{C}$
for each block $P_{\mathcal{C}} \in \mathcal{C}$ **do**
  if $s = \emptyset \parallel P_{\mathcal{C}} \neq P_s$ **then**
    $V_{\mathcal{C}} \leftarrow P_{\mathcal{C}} \cap \mathcal{C}$
    for each block $P \notin \mathcal{C}$ s.t. $|\{N_P(V_{\mathcal{C}}) \setminus \{s\}\}| > 1$ **do**
      $V_{\mathcal{C}}^{'} \leftarrow V_{\mathcal{C}}$, $\mathcal{E}_P(V_{\mathcal{C}}^{'}) \leftarrow \emptyset$ // Goal is to build upon $\mathcal{E}_P(V_{\mathcal{C}}^{'})$
      $\mathcal{H} \leftarrow \{N_P(V_{\mathcal{C}}) \setminus \{s\}\}$ max-heapified by adjacency count to $V_{\mathcal{C}}$
      `MIN_COVER` $\leftarrow$ `CUR_COVER` $\leftarrow \omega(\mathcal{C})$
      while $|\mathcal{H}| > 0$ and `CUR_COVER` $\geq$ `MIN_COVER` **do**
        $n \leftarrow$ `pop-heap`$(\mathcal{H})$
        `CUR_COVER` $\leftarrow \omega(\{\mathcal{C} \setminus V_{\mathcal{C}}\} \cup \{V_{\mathcal{C}}^{'} \cap N(n)\} \cup \{\mathcal{E}_P(V_{\mathcal{C}}^{'}) \cup n\})$
        if `CUR_COVER` $\geq$ `MIN_COVER` **then**
          $V_{\mathcal{C}}^{'} \leftarrow V_{\mathcal{C}}^{'} \cap N(n)$
          $\mathcal{E}_P(V_{\mathcal{C}}^{'}) \leftarrow \mathcal{E}_P(V_{\mathcal{C}}^{'}) \cup n$
          `MIN_COVER` $\leftarrow$ `CUR_COVER`
        end if
      end while
      if `CUR_COVER` $> \omega(\hat{\mathcal{C}})$ **then**
        $\hat{\mathcal{C}} \leftarrow \{\mathcal{C} \setminus V_{\mathcal{C}}\} \cup V_{\mathcal{C}}^{'} \cup \mathcal{E}_P(V_{\mathcal{C}}^{'})$
      end if
    end for
  end if
end for
return $\hat{\mathcal{C}}$

This algorithm can be computed trivially and an implementation allowing efficient updating using a Fibonacci heap is described in [52]. We omit its algorithmic definition, because we also explore the use of a random seeding method in Section 4.3.1. We will also present variations on the algorithms in Section 4.2.2.

To demonstrate the procedure of the algorithm, Figure 4.3 presents an example graph. Here, nodes with numeric labels 1–4 indicate the order in which nodes should be added to the optimal multipartite clique containing black labeled nodes and edges, where the seed node s is the node labeled 1. The blue nodes labeled 2 represent nodes that are part of the clique initially, but are removed to enable the growth of a larger clique. Specifically, all nodes labeled 1 and 2 correspond to the state of the clique at line 2 in Algorithm 3, where the nodes labeled 2 correspond to $\mathcal{E}_P(s)$. When Algorithm 4 is called in line 3, extensions from P_2 to P_1 or P_3 are considered. Blue and black edges represent edges that will be visited, while green edges will not. The adjacency counts for each neighboring node are displayed, corresponding to the contents of the max-heap $\mathcal{H}$ from line 7 of Algorithm 4. Ideally, the nodes labeled 3 from P_1 would be added to $\mathcal{C}$ first, followed by the nodes labeled 4 from P_3. However, the algorithm would choose the nodes labeled 4 over the nodes labeled 3 because the coverage is better. This motivates one of the variations of our algorithm that will be described in Section 4.2.2.

We allow the iterations to continue until the seed node does not neighbor any block for which it contains more than one edge to (see line 3 of Algorithm 2). In this case, the best $1 \times N$ biclique that could be built from s would be a single edge, and our algorithm could not offer further compression of the graph. We assume that the remaining uncovered edges would exist only in multipartite cliques that contain a single node per block. It is possible at this point that more cliques could exist, but they would have to be small and our experiments indicate that this stopping point does not occur prematurely. The results of our algorithm's performance on the CMGS database graph will be presented in Section 4.3.

4.2.2 Variations

In order to validate our algorithmic choices, we developed variations of the search strategy for building a multipartite clique from a seed node to compare against. In particular, we identified three areas of the search strategy that could be designed differently and might have an effect on the overall quality of our clustering. One such variation was alluded to in Section 4.2.1 with respect to Figure 4.3. Our results implemented with these variations are presented in Section 4.3. In addition, our choice of the seed node is discussed in Section 4.3.1. In this section, we explain the three sources for variation in the search strategy for our algorithm.

1. **Required backtracking:** As described previously with respect to Figure 4.3, it is possible that desirable nodes from the seed node's block P_s will be excluded if at line 3

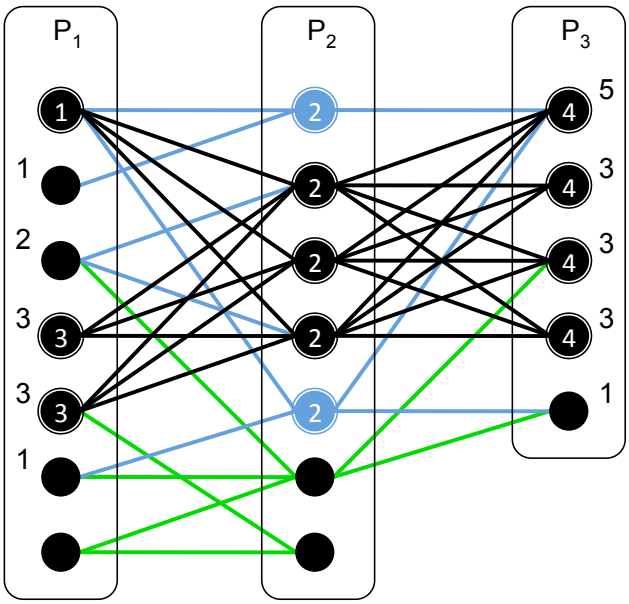


Figure 4.3: Shows the process by which nodes are added to build a multipartite clique (black edges) from a seed node s (labeled 1). Blue edges are explored but not involved in the final multipartite clique. Green edges are not explored. Numeric labels and colors of nodes indicate the order in which nodes are added and removed in the process of building the multipartite clique.

in Algorithm 3, an extension is made to a new block instead. If, however, the algorithm is required to try to extend to P_s first, we would get the desired output. We refer to this behavior as *backtracking* because we backtrack to the seed node's block, and we have implemented a version of the algorithm where backtracking is required. In the presented example, it is clear that required backtracking is advantageous, but we could easily produce examples where it is not. We study the effect of requiring versus only allowing backtracking and present the results in Section 4.3.

2. **Allowing $1 \times N$ bicliques:** Although theoretically permissible, in some applications it may be undesirable to present bicliques of size $\{1 \times N\}$. Because line 3 of Algorithm 4 requires that the initial $\{1 \times N\}$ biclique containing s and $\mathcal{E}_P(s)$ not be extended about block P_s, there is a limited set of extensions that can occur initially. The algorithm presented does not require an extension be made at line 3 of Algorithm 3, but we have also implemented a version that requires an extension be made initially to prevent $\{1 \times N\}$ bicliques from being returned.

3. **Strict checking of $1 \times N$ bicliques:** In the variation described above, we explain how in order to discourage the reporting of $1 \times N$ bicliques, we require an initial extension be made with Algorithm 4 during the call from line 3 in Algorithm 3. However, Figure 4.4 shows how even with this requirement, some $1 \times N$ bicliques could still occur if the extension set becomes reduced to a single node. In this example, backtracking adds two nodes labeled 3 from P_1, but throws out four of the five nodes from the extension set in P_2 (the blue nodes labeled 2) and would end up reporting a smaller biclique. A similar scenario could occur if the extension is not made via backtracking. Therefore, we also implemented a stricter policy that detects these scenarios during the initial extension, in which case we keep the initial $1 \times N$ biclique produced by s and the extension set. This prevents reporting a smaller multipartite clique due to the stubbornness of the algorithm in trying to make an initial extension.

4.3 Results

In this section, we present the results of running our algorithm with and without the variations described in Section 4.2.2. As we expected, allowing for $1 \times N$ bicliques to be reported produced the best multipartite clique cover. On the other hand, we did not find any significant difference whether or not we required backtracking; but through manual qualitative analysis of the first few reported clusters, requiring backtracking produced clusters that made more sense biologically. This will be discussed in Section 4.3.3. In the following results, we refer to each experiment by which variation options were used. `No opts` corresponds to the standard algorithm with no variations (requires initial extension and allows, but does not require, backtracking). The `R` option implies required backtracking. The `O` option corresponds to allowing $1 \times N$ bicliques, which translates to not requiring an initial extension.

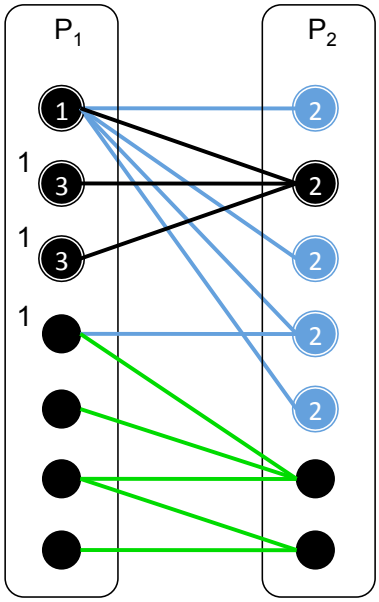


Figure 4.4: Shows how requiring an initial extension (black nodes labeled 3) could still produce a $1 \times N$ biclique (black edges) that is smaller than that produced with just the seed node (labeled 1) and extension set (nodes labeled 2).

Lastly, the strict checking of $1 \times N$ bicliques variation is denoted either `(strict)` or with the appending of `_s`.

Figures 4.5 and 4.6 display the quality of the multipartite clique cover produced with each version of the algorithm. In particular, Figure 4.5 shows the number of multipartite clique clusters produced by each method and the distribution of the number of blocks contained in each cluster. Figure 4.6 compares the coverage in terms of percentage of edges in the input graph covered by the clusters (suggesting how close it gets to a complete multipartite clique cover without considering singletons) versus the average edge content of each cluster. For example, Figure 4.6 shows that the standard `No opts` algorithm produces the best coverage of the input graph and using variations `RO` and `O` alone produce the smallest coverage, but all methods produce roughly the same edge coverage ($\sim$90%). However, looking at average edge content of each cluster and the number of clusters reported by each method in Figure 4.5, it is clear that `RO` and `O` achieve almost the same coverage but with fewer, larger clusters. Thus, we observe that the multipartite clique cover produced by allowing for $1 \times N$ bicliques is the best, and allowing versus requiring backtracking does not appear to help or hurt the method. If, on the other hand, it is desirable to find multipartite cliques that are not $1 \times N$ bicliques, thus not using the `O` option, we observe that both the variations for required backtracking and strict checking of $1 \times N$ tend to offer better multipartite clique covers.

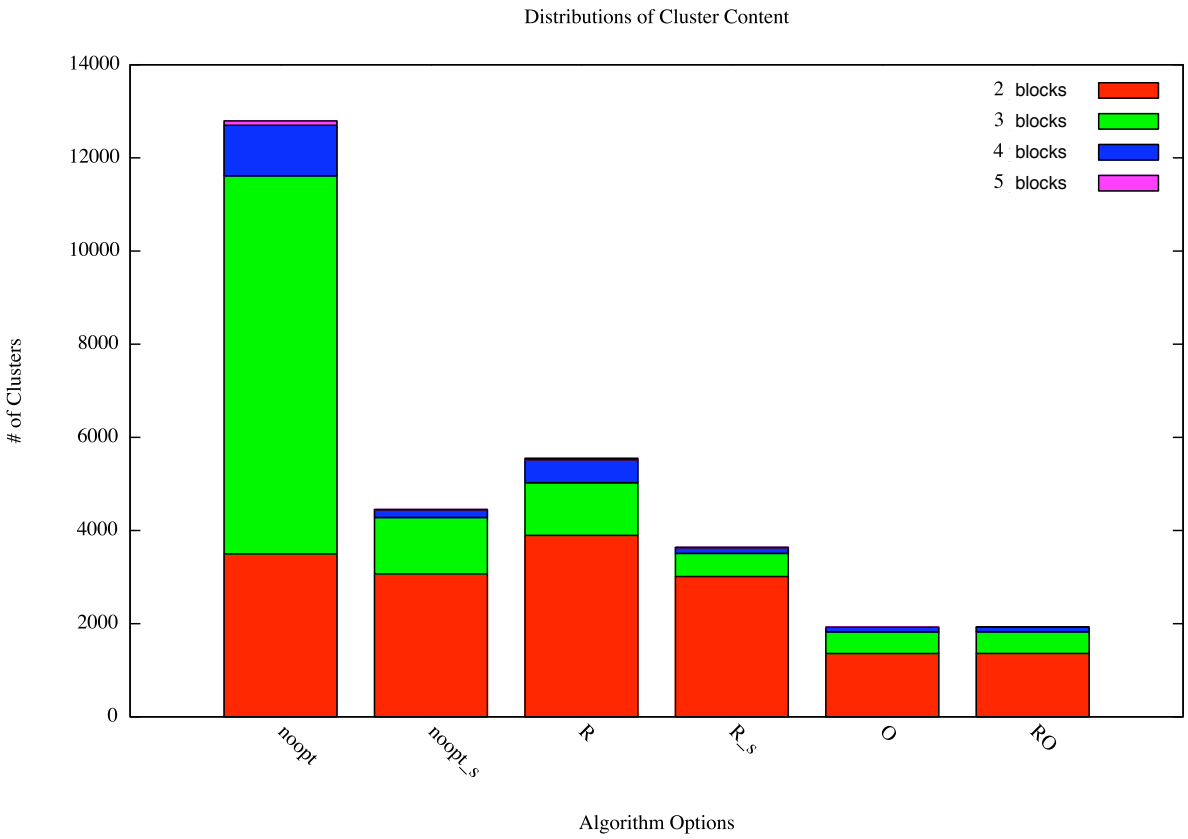


Figure 4.5: Number and distribution of clusters returned with different variations.

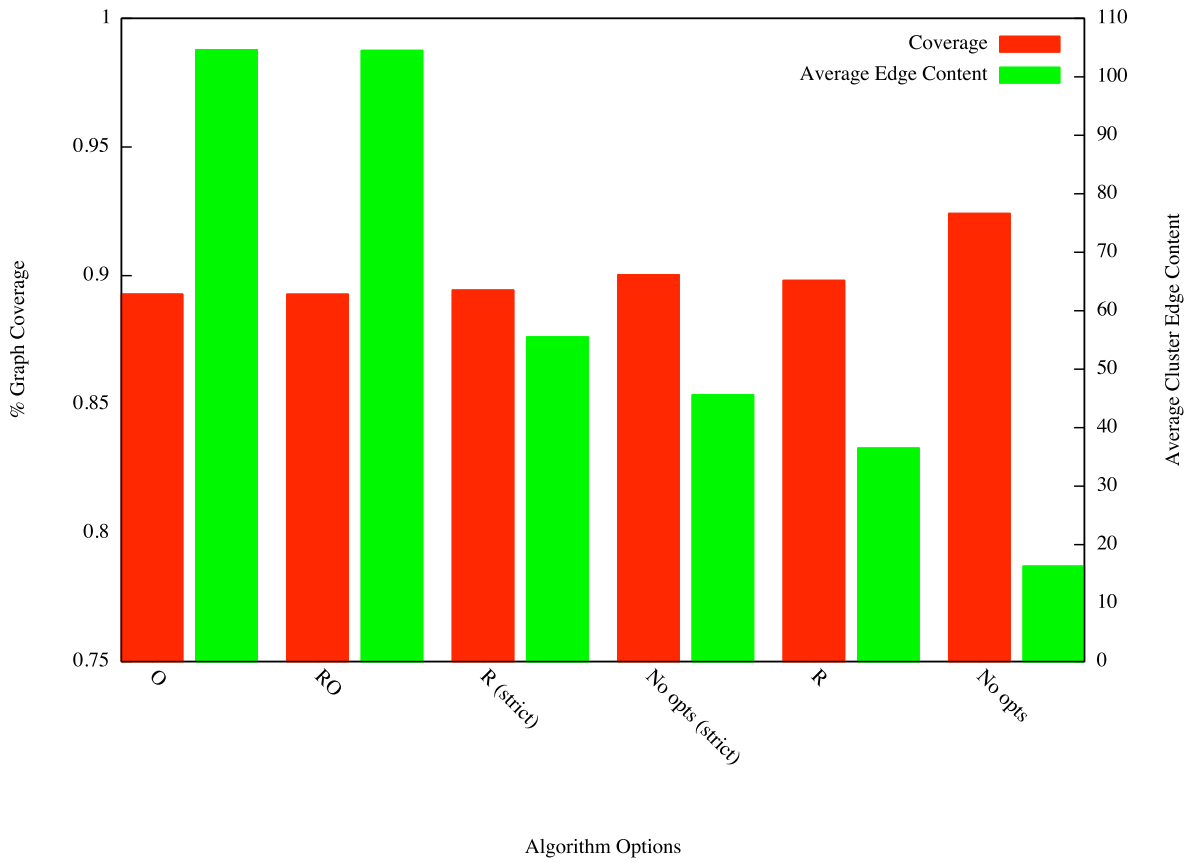


Figure 4.6: Percent edge coverage of graph by clusters returned from each variation plotted alongside the average edge coverage per cluster.

4.3.1 Seed Selection

Next, we investigate the effectiveness of our seed selection method where, at each iteration, multipartite cliques are built from a seed node with the largest degree (see Section 4.2.1). We compare our algorithm using the largest degree seed node (LDSN) versus a random seed node (RSN) to see if the resulting multipartite clique covers differ in quality. We use the `RO` version of the algorithm because as shown in our results from Section 4.3, this variation provides one of the best multipartite clique covers. We choose this over version `O` because requiring backtracking has shown to be beneficial if $1 \times N$ bicliques are discouraged.

To fairly compare the LDSN selection method to the RSN selection method, we recognize that the stopping condition of Algorithm 2 at line 3 would occur frequently and produce a very poor multipartite clique cover. To overcome this bias, at each iteration, we repetitively select seed nodes (line 2) until we pick a seed that would not trigger the stopping condition on line 3. To allow unbiased comparison of seed node selection schemes and prevent infinite looping, we allow the method to generate the same number of clusters generated with LDSN selection (1926) and use this as a stopping condition.

Figures 4.7 and 4.8 compare each run using the `RO` algorithm and RSN selection with the corresponding LDSN selection results presented in Figures 4.5 and 4.6. Most importantly, Figure 4.8 shows that the overall edge coverage of the clusters as well as average cluster edge content using RSN selection tend to be smaller than when using the LDSN selection method we proposed. Figure 4.5 reveals an interesting trend as well. Using LDSN selection produces a smaller percentage of bicliques than with RSN selection and a larger percentage of 3-partite cliques, but RSN selection produces more 5/6-partite cliques.

To explore these observations more carefully, we looked at the cluster size distributions in terms of edge coverage (Figure 4.9) as well as the distributions of the bicliques produced by each method (Figure 4.10). With the size distribution, the clusters produced by both seed selection methods follow a similar size distribution, indicating that they do not produce dramatically different multipartite clique covers. However, LDSN selection produces noticeably more clusters in the largest size category than RSN selection and fewer clusters in the smaller size categories between 2 and 200 edges. This explains why the multipartite clique cover produced with LDSN selection was better than that produced with RSN selection.

Next, we were interested in understanding how each seed selection method performs in selecting bicliques since they are the most common type of multipartite clique reported (see Figures 4.5 and 4.7). Specifically, we wanted to understand if using LDSN selection tends to extract larger cliques at earlier iterations when the graph is more dense, which would indicate that it works better than RSN selection for extracting large multipartite cliques in general (i.e., not for producing a multipartite clique cover). Figure 4.10 shows the comparison of the discovered bicliques' size distributions, where a biclique's size is shown as being $N \times M$ with $N \leq M$. In these two figures, the different data point styles indicate the order in which the bicliques are discovered. For example, the data points corresponding to the series 1-100 are

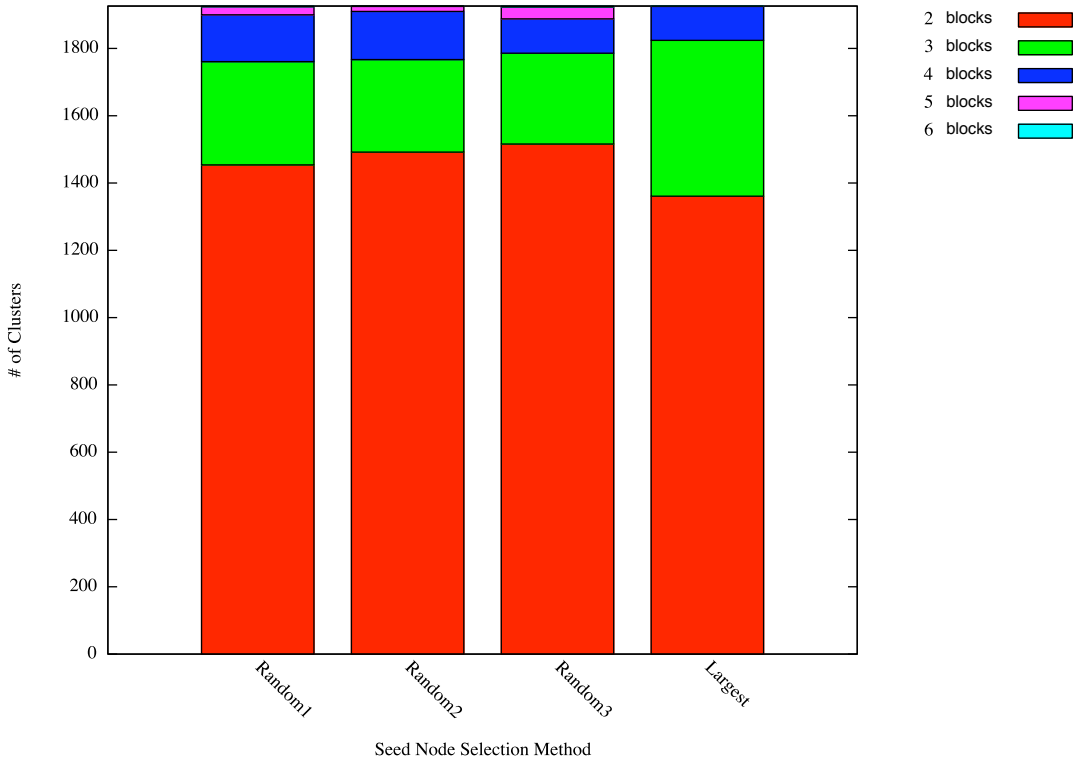


Figure 4.7: Number and distribution of clusters returned with `RO` algorithm variations with LDSN selection (Largest) and RSN selection (Random).

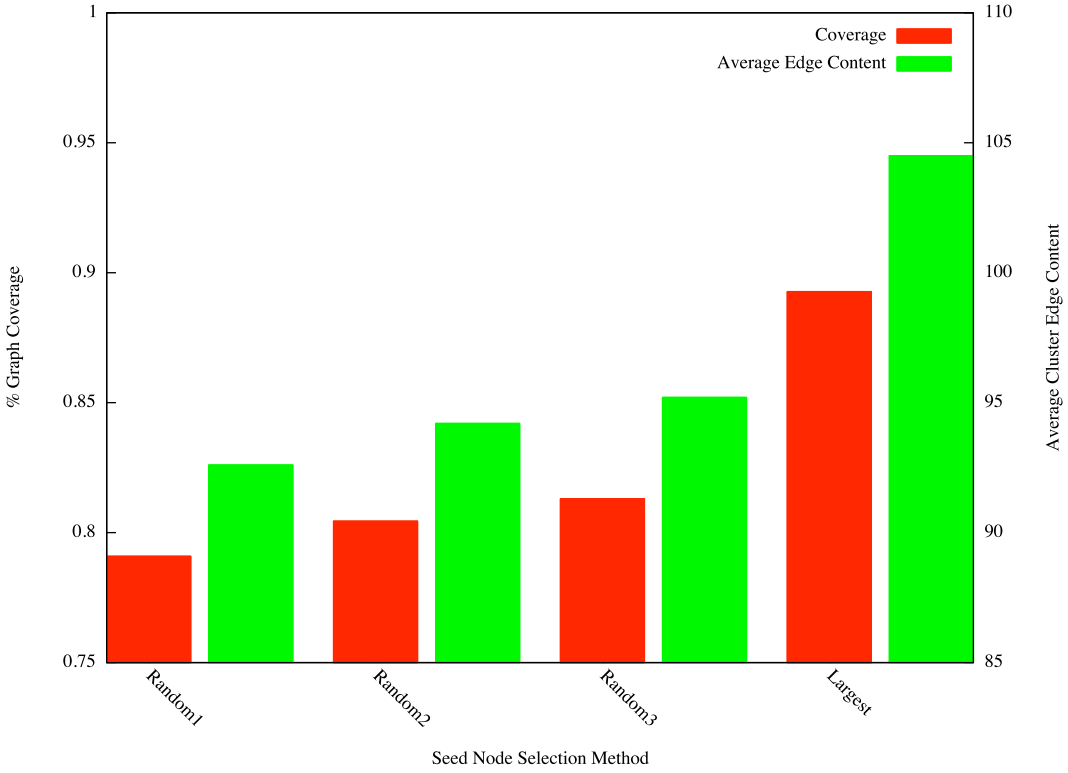


Figure 4.8: Percent edge coverage of graph by clusters returned with `RO` algorithm variations with LDSN selection (Largest) and RSN selection (Random) alongside the average edge coverage per cluster.

the first 100 bicliques discovered by the algorithm. There was no particular significance to the method for selecting bin sizes, the bin size simply doubles as it grows. These distributions tell us that using LDSN selection methodically picks the largest bicliques from the input graph while it is more dense. On the contrary, RSN selection does not tend to find larger bicliques first. This may explain why it produces smaller multipartite cliques in general, because picking a bad seed node may cause the algorithm to discover suboptimal multipartite cliques more frequently. These results imply that our LDSN selection method works well and in turn provides a significantly better approximation of the optimal multipartite clique cover than building multipartite cliques from random positions in the graph.

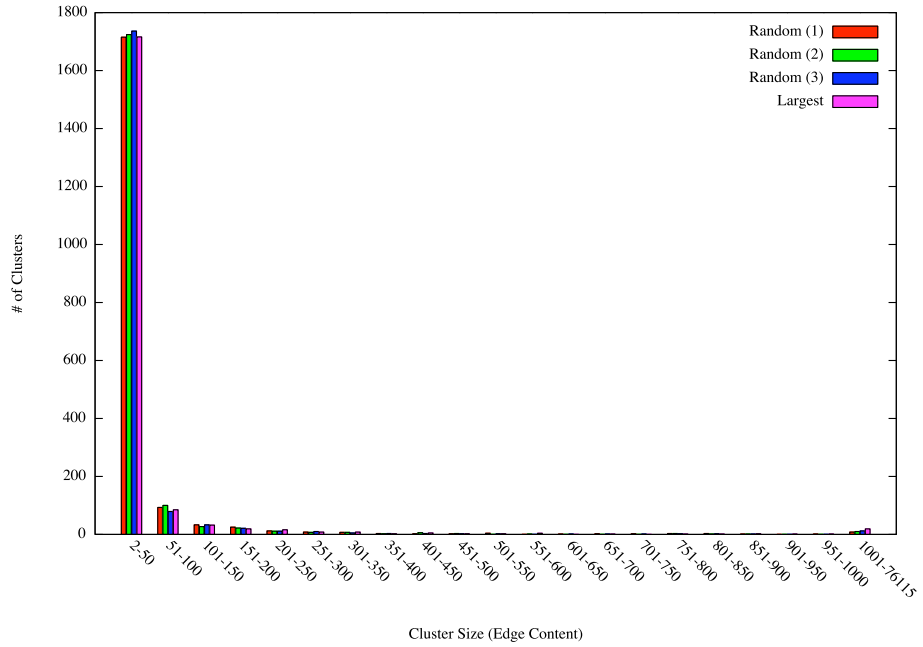
4.3.2 Closure of Multipartite Cliques

Figure 4.1 shows that our multipartite graph may contain cycles among partition blocks; namely between genes, phenotypes, and GO categories. This cyclic behavior is common in multipartite graphs. Our algorithm, however, only allows for addition of new blocks at each iteration (see line 5 in Algorithm 4). This restricts a multipartite clique to contain only $n-1$ of the possible n edge types within an n-block cycle. We implemented a *closure* step for post-processing to handle these cases. Algorithm 5 presents this method, using the notation (P_i, P_j) to indicate adjacency between blocks P_i and P_j, and E is the set of edge type adjacencies. Note we do not require edges in the closure to form a clique. Rather, we report the density $\mathcal{D}$ (see Section 4.1.2) of the edges in the closure.

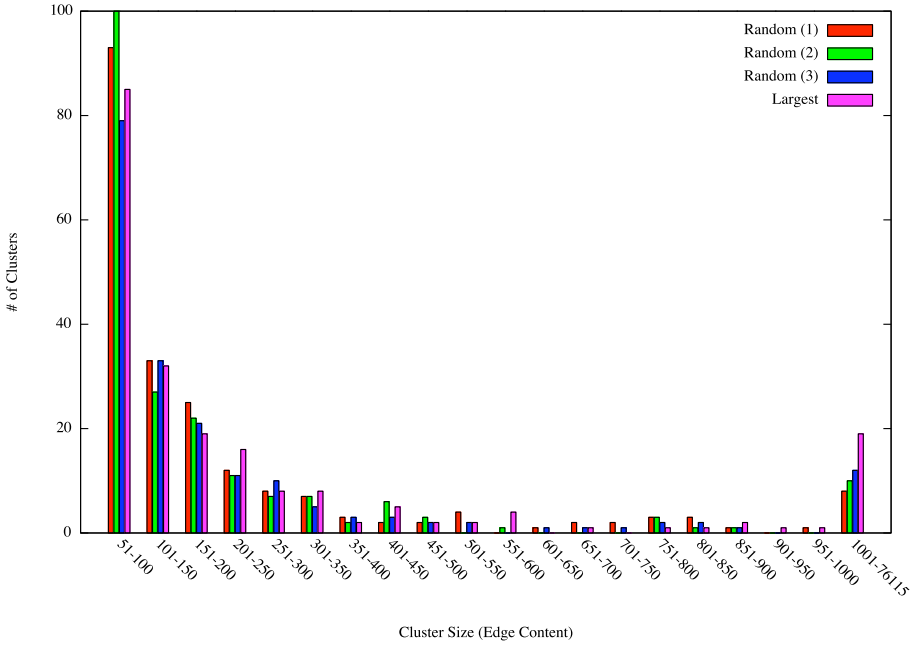
We found that the closure step does not have a significant effect on the results, mainly due to the infrequency of cycles in the input graph and the low edge density among the only cycle. We found that only 95 edges were picked up with closure, which corresponds to $\mathcal{D} =$ 7.12E-3. However, this density is larger than the expected edge density of the edges within the cycle ($\mathcal{D} =$ 1.53E-3). Further, compared to the `RO` run with LDSN selection presented in Section 4.3, using closure produced a multipartite clique cover with 1770 clusters covering 88.97% of the input graph compared to 1926 clusters covering 89.28% of the input graph without closure. Thus the coverage was slightly smaller, but the average cluster size (edge coverage) increased from 104.5 to 113.4. We expect that although closure did not generate a significantly different clustering, in multipartite graphs containing more cyclic partition blocks, it would be a necessary step.

4.3.3 Biological Analysis of Clusters

The CMGS database that we cluster contains two phenotypes `WBPhen336` and `PBPhen1`, both of which correspond to wild-type phenotypes. `WBPhen336` also corresponds to the first node selected by LDSN selection (i.e., $|N(\texttt{WBPhen336})|$ is maximal). The popularity of these phenotypes is due to the majority of genes whose knockdown does not cause noticeable abnormalities. We expected that the biclique between these two phenotypes and ~ 17000

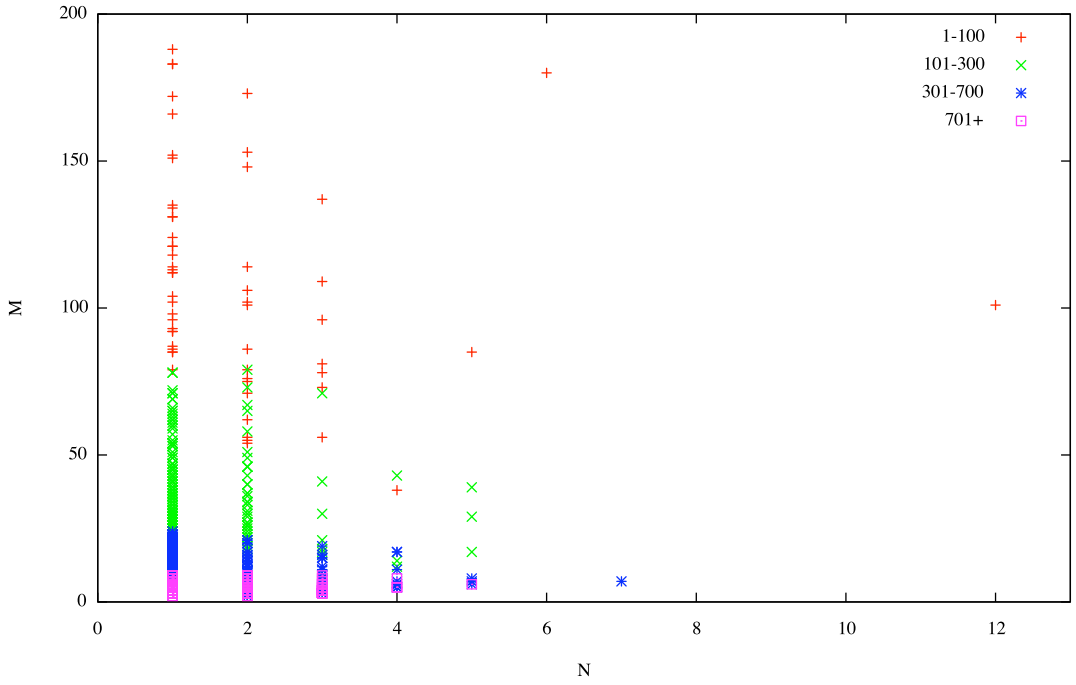


(a) Full distribution.

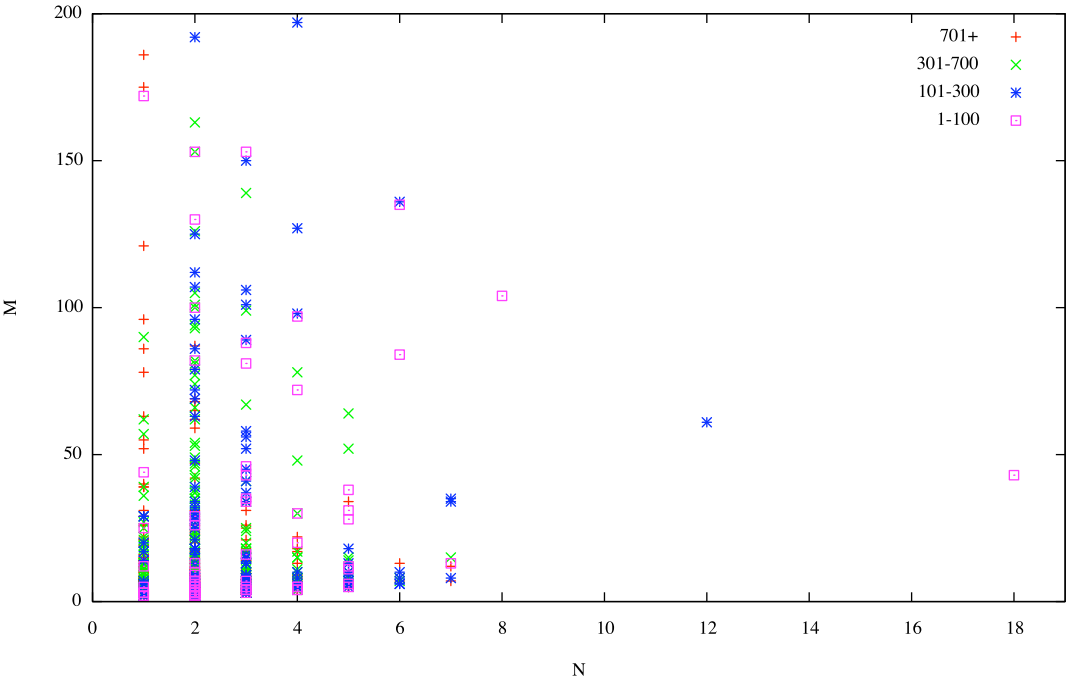


(b) Distribution of clusters with >50 edges.

Figure 4.9: Histogram of size (edge coverage) distribution of clusters using RSN selection (Random) and LDSN selection (Largest).



(a) LDSN selection.



(b) RSN selection.

Figure 4.10: Distribution of bicliques of size $N \times M$ between LDSN selection and RSN selection (zoomed on M $\in$ [0,200]) displayed in the order in which they were discovered by the algorithm.

Algorithm 5 CLOSECLIQUE($\mathcal{MG}, \mathcal{C}$):

Input: A multipartite graph $\mathcal{MG}$ and a clique $\mathcal{C}$.
Output: The closed version of $\mathcal{C}$, and a set $\mathbb{D}$ of closure densities.

$\mathbb{D} \leftarrow \emptyset$
for all pairs of partition blocks $P_i, P_j \in \mathcal{C}$ **do**
  if $(P_i, P_j) \in E^{\mathcal{MG}}$ and $(P_i, P_j) \notin E^{\mathcal{C}}$ **then**
    $V_i \leftarrow P_i \cap \mathcal{C}$
    $V_j \leftarrow P_j \cap \mathcal{C}$
    $D \leftarrow 0$
    for all possible edges $(i, j) \in \{V_i \times V_j\}$ **do**
      if $(i, j) \in E^{\mathcal{MG}}$ **then**
        $\mathcal{C} \leftarrow \mathcal{C} \cup (i, j)$
        $D \leftarrow D + 1$
      end if
    end for
    $\mathcal{D}_{ij} \leftarrow \frac{D}{|V_i| \cdot |V_j|}$
    $\mathbb{D} \leftarrow \mathbb{D} \cup \mathcal{D}_{ij}$
  end if
end for
return $\mathcal{C}, \mathbb{D}$

genes was the largest in the graph, so we used this as a benchmark.

With LDSN selection and `RO` options, the first cluster recognized contained $>$ 15000 genes whose knockdown was related to these two wild-type phenotypes as well as contained orthologs in 3 species: *C. elegans*, *C. briggsae*, and *C. remanei*. These species are the most genetically similar amongst all species in the database. When using just the `O` option, $<$ 12000 genes were selected with connectivity to only the one wild-type phenotype `WBPhen336` and contained orthologs in 5 species. This is still a legitimate cluster, but we would expect to see both wild-type phenotypes, suggesting that required backtracking creates better clusters. With RSN selection, the wild-type phenotypes were spread across many more clusters than with LDSN selection.

Below, two example clusters chosen randomly that represent biologically meaningful relationships are displayed:

- Genes
  - pmk-1: encodes a mitogen-activated protein kinase (MAPK), orthologous to human p38 MAPK (OMIM:600289), that is required for eliciting gonadal programmed cell death in response to Salmonella enterica infection; PMK-1 lies upstream of CED-9, a negative regulator of apoptosis, in the programmed cell death pathway.

- GO Categories
  - GO:0006972: hyperosmotic response
  - GO:0012501: programmed cell death
  - GO:0045087: innate immune response
- Phenotypes
  - WBPhen35: Enhanced susceptibility to pathogens
- Proteins
  - CE06686: Serine/threonine kinase, encoded by pmk-1
  - CE40030: Protein interaction
  - CE39588: Protein interaction
  - CE37741: Protein interaction
  - ... 33 other protein interactions
- Adjacency:

  Genes ↔ Phenotypes

  Genes ↔ GO Categories

  Genes ↔ Proteins

In this cluster, the gene pmk-1, which is annotated as having relationships to apoptosis (programmed cell death), is related to meaningful GO categorizations and phenotypes. It also contains a set of proteins that pmk-1 encodes or interacts with.

- Genes
  - gei-4: predicted to mediate protein-protein interactions
- GO Categories
  - GO:0005200: structural constituent of cytoskeleton
  - GO:0005882: intermediate filament
  - GO:0009790: embryonic development
  - GO:0045109: intermediate filament organization
- Phenotypes
  - MultiVulvaePhen: growth of multiple vulvae

  - OsmosensePhen1: Osmoprotection (greatly reduced glycerol accumulation, brood size, fertility, growth rate under hypertonic stress)
  - WBPhen209: larval lethal
  - WildTypeVulvaPhen: single vulva as in wild-type

- Proteins
  - CE40377
  - CE40362
  - CE40240
  - ... $\sim$300 other protein interactions

- Adjacency:

  Genes $\leftrightarrow$ Phenotypes

  Genes $\leftrightarrow$ GO Categories

  Genes $\leftrightarrow$ Proteins

In this cluster, the gene gei-4 is annotated as likely to mediate protein-protein interactions, and its knockdown causes many abnormal phenotypes including lethality. This would not be surprising given the significance of the gene implied by its GO categorizations as well as the large number of proteins it interacts with.

The clusters we have presented are only two from a set of $\sim$2000. These are merely meant to suggest the biological significance of our clusters and that through minimal inspection, it can be seen that the relationships are meaningful and may be used to make implications based on the multiple sources of data integrated into a single cluster.

4.4 Related Work

Traditional clustering algorithms like k-means [22] work by forming clusters through minimization of a distance metric between data objects that contain different values defined over the same set of attributes (`x,y,z` coordinates for example). Obvious metrics include, but are not limited to, Euclidean distance and correlation coefficients that have sound statistical significance. Clustering is common in high-throughput biological data analysis such as microarray gene expression analysis, where different experimental conditions provide a vector of expression measurements corresponding to individual genes; a survey on various techniques is presented in [26]. Clusters of genes therefore represent genes that exhibit similar expression patterns in response to various stimuli.

Graph clustering is a well studied subtopic of clustering due to the prevalence of graph datasets and their complex nature. In general graphs, the adjacency matrix can be viewed as the fixed set of attributes implying a transformation allowing the use of a traditional clustering method. However, graphs can exhibit other properties that make traditional clustering methods unsuitable. Multipartite graphs provide an example, since connectivity between certain blocks of nodes is interesting, not all nodes. Multipartite graphs are referred to as *k-partite* graphs in [35] where a relation summary network (RSN) is generated from the graph and is clustered with k-means using various graph divergence measures. The drawback of this method is that k-means requires an input parameter specifying the desired number of clusters and analysis is only performed on graphs with up to three partition blocks containing no cyclic behavior among blocks. Many methods for graph clustering rely on edge cut properties [15] and do not allow overlap between nodes or edges in different clusters.

Multipartite graph clustering, particularly in finding maximal multipartite cliques, has been studied in [52] as a method for detecting orthologous genes in various species. This method differs from ours due to the existence of edge weights (based on gene sequence similarity) and adjacency between all blocks.

Chapter 5

Conclusions

In this thesis, we have addressed two topics in graph mining. In the first application, we have presented a general and expressive framework for diagnosing memory leaks using frequent graph grammar mining. Our work extends the arsenal of memory leak diagnosis tools available to software developers. Using sampling schemes and summarization of garbage collection root paths, we improve the scalability and contextual expressiveness of graph grammar mining. We have introduced the notion of dominators for graph mining and how they possess sufficient statistics for discovering certain types of frequent subgraphs. The experimental results are promising in their potential to debug leaks when other state-of-the-art tools cannot.

Our future work on this topic revolves around three themes. First, we seek to embed our algorithm in a runtime infrastructure so that it can track leaking subgraphs as they build up over time. Second, we seek to investigate the theoretical properties of dominators and whether they can support a frequent pattern growth [20] style of subgraph mining. This approach will allow us to process larger heap dumps than our current approach. Third, we plan to perform a quantitative evaluation comparing the quality of our reports to existing tools. Such quantitative comparisons require the definition of a metric, which could be derived by approximating the number of lines of code a user would have to investigate to verify the presence or absence of a bug, as proposed in [46].

In another application, this thesis also has addressed the topic of multipartite graph clustering on a heterogeneous biological database. We have presented an algorithm for approximation of the optimal multipartite clique cover and demonstrated its effectiveness as a database summarization method. This graph mining technique provides a novel algorithm for analysis of large multipartite graphs and is directly extendible to many relational databases, both biological and otherwise. We have shown that using LDSN selection provides a simple, high quality search heuristic. When applied to an integrated *C. elegans* database, we have exhibited its ability to improve the comprehension of massive quantities of data as well as to extract large clusters of biots that are shown to be biologically meaningful.

Areas for future work from this study stem mainly from our lack of comparative analysis of our algorithm. We would like to test our algorithm on other databases and random multipartite graphs to ensure that bias does not exist among the evaluation of our algorithmic choices due to the structure of our input database. A more thorough comparison of the quality of our clusters may also prove beneficial. In terms of cluster content, it would be useful to compare our results to the equivalent of a gold-standard manually curated set of clusters, similar in concept to the use of the eukaryotic orthologous groups (KOG) database for comparison in [52]. However, these integrated databases are relatively new and not extensively studied, so we do not expect that such a reference dataset currently exists. In terms of the overall clustering quality of our algorithm, we seek to compare the clustering produced by our algorithm to other works in database clustering, such as [35]. These works do not necessarily assume the same graph structure [52] and may look for clusters that satisfy different properties, such as clusters that allow edge overlap or disallow node overlap [35] further complicating a comparison.

Bibliography

[1] Bug 38048: Classloader leak caused by EL evaluation, Oct. 2006. https://issues.apache.org/bugzilla/show_bug.cgi?id=38048a.

[2] Computational models for gene silencing, 2008. https://bioinformatics.cs.vt.edu/cmgs/CMGSDB/.

[3] Wormbase web site, Apr. 2010. http://www.wormbase.org/.

[4] M. Akbar and R. Angryk. Frequent pattern-growth approach for document organization. In *CIKM '08*, pages 77–82, 2008.

[5] G. D. Bader and C. W. V. Hogue. An automated method for finding molecular complexes in large protein interaction networks. *BMC Bioinformatics*, 4:2, 2003.

[6] A. Bailey and G. Back. LibX–A Firefox extension for enhanced library access. *Library Hi Tech*, 24(2):290–304, 2006.

[7] M. Bond and K. McKinley. Tolerating memory leaks. In *OOPSLA '08*, pages 109–126, 2008.

[8] M. D. Bond and K. S. McKinley. Bell: Bit-encoding online memory leak detection. In *ASPLOS-XII '06*, pages 61–72, 2006.

[9] T.A. Brown. *Genomes 3*. Garland Science, third edition, May 2006.

[10] H. Cheng, D. Lo, Y. Zhou, X. Wang, and X. Yan. Identifying bug signatures using discriminative graph mining. In *ISSTA '09*, pages 141–152, New York, NY, USA, 2009. ACM.

[11] C. Chent, X. Yan, F. Zhu, and J. Han. gApprox: Mining frequent approximate patterns from a massive network. In *ICDM '07*, pages 445–450, 2007.

[12] S. Cherem, L. Princehouse, and R. Rugina. Practical memory leak detection using guarded value-flow analysis. In *PLDI '07*, pages 480–491, 2007.

[13] D. Cook and L. Holder. Substructure discovery using minimum description length and background knowledge. *JAIR*, 1:231–255, 1994.

[14] M. Dawande, P. Keskinocak, J. M. Swaminathan, and S. Tayur. On bipartite and multipartite clique problems. *Journal of Algorithms*, 41(2):388–403, 2001.

[15] C. H. Q. Ding, X. He, et al. A min-max cut algorithm for graph partitioning and data clustering. In *ICDM '01*, pages 107–114, 2001.

[16] J. Engelfriet and G. Rozenberg. Graph grammars based on node rewriting: An introduction to NLC graph grammars. In *Graph grammars and their application to computer science: 4th International Workshop, Proceedings.*, pages 12–23, 1991.

[17] S. Fields and O.K. Song. A novel genetic system to detect protein-protein interactions. *Nature*, 340:245–246, July 1989.

[18] A. Fire, S. Xu, et al. Potent and specific genetic interference by double-stranded RNA in *Caenorhabditis elegans*. *Nature*, 391:806–811, February 1998.

[19] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[20] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *SIGMOD '00*, pages 1–12, 2000.

[21] T. W. Harris, N. Chen, F. Cunningham, et al. Wormbase: a multi-species resource for nematode biology and genomics. *Nucleic Acids Res*, 32(Database issue):D411–D417, 2004.

[22] J. A. Hartigan and M. A. Wong. A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):100–108, 1979.

[23] R. Hastings and B. Joyce. Purify: A tool for detecting memory leaks and access errors in C and C++ programs. In *Winter USENIX Conference*, pages 125–138, 1992.

[24] M. Hauswirth and T. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *ASPLOS-XI '04*, pages 156–164, 2004.

[25] D. Heine and M. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *PLDI '03*, pages 168–181, 2003.

[26] D. Jiang, C. Tang, and A. Zhang. Cluster analysis for gene expression data: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 16:1370–1386, 2004.

[27] I. Jonyer, L. Holder, and D. Cook. MDL-based context-free graph grammar induction and applications. *IJAIT*, 13(1):65–79, 2004.

[28] M. Jump and K. McKinley. Cork: Dynamic memory leak detection for garbage-collected languages. In *POPL '07*, pages 31–38, 2007.

[29] J. Kukluk, L. Holder, and D. Cook. Inference of node and edge replacement graph grammars. In *Proc. ICML Grammar Induction Workshop*, 2007.

[30] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *ICDM '01*, pages 313–320, 2001.

[31] M. Kuramochi and G. Karypis. Discovering frequent geometric subgraphs. *Inf. Syst.*, 32(8):1101–1120, 2007.

[32] D.A. Lashkari, J.L. DeRisi, et al. Yeast microarrays for genome wide parallel genetic and gene expression analysis. *PNAS*, 94(24):13057–62, 1997.

[33] T. Lengauer and R. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, 1979.

[34] C. Liu, X. Yan, H. Yu, J. Han, and P. S. Yu. Mining behavior graphs for "backtrace" of noncrashing bugs. In *SDM '05*, pages 286–297.

[35] B. Long, X. Wu, Z. Zhang, and P. S. Yu. Unsupervised learning on k-partite graphs. In *KDD '06*, pages 317–326, 2006.

[36] E. Maxwell, G. Back, and N. Ramakrishnan. Diagnosing memory leaks using graph mining on heap dumps. In *KDD '10 (**to appear**)*, July 2010.

[37] N. Mitchell. The runtime structure of object ownership. In *ECOOP '06*, pages 74–98, 2006.

[38] N. Mitchell and G. Sevitsky. LeakBot: An automated and lightweight tool for diagnosing memory leaks in large Java applications. In *ECOOP '03*, pages 151–172, 2003.

[39] N. Mitchell and G. Sevitsky. The causes of bloat, the limits of health. In *OOPSLA '07*, pages 245–260, 2007.

[40] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI '07*, pages 89–100, 2007.

[41] M. Orlovich and R. Rugina. Memory leak analysis by contradiction. In *Lecture Notes in Computer Science*, volume 4134, pages 405–424. Springer, 2006.

[42] A. Pati, Y. Jin, K. Klage, R. F. Helm, L. S. Heath, and N. Ramakrishnan. CMGSDB: Integrating heterogeneous *Caenorhabditis elegans* data sources using compositional data mining. *Nucleic Acids Research*, 36(Database-Issue):69–76, 2008.

[43] W. De Pauw and G. Sevitsky. Visualizing reference patterns for solving memory leaks in Java. *Concurrency—Practice and Experience*, 12(14):1431–1454, 2000.

[44] R. Peeters. The maximum edge biclique problem is NP-complete. *Discrete Appl. Math.*, 131(3):651–654, 2003.

[45] B. Ren, F. Robert, et al. Genome-wide location and function of DNA binding proteins. *Science*, 290(5500):2306–9, 2000.

[46] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *ASE '03*, pages 30–39, 2003.

[47] J. Rissanen. *Stochastic Complexity in Statistical Inquiry Theory*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1989.

[48] G. Rozenberg. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1. World Scientific, 1997.

[49] S.C. Schuster. Next-generation sequencing transforms today's biology. *Nature Methods*, 5(1):16–18, 2008.

[50] J. Siek, L. Lee, and A. Lumsdaine. Boost graph library, June 2000. http://www.boost.org/libs/graph/.

[51] E. Tilevich and G. Back. Program, enhance thyself! Demand-driven pattern-oriented program enhancement. In *AOSD '08*, pages 13–24, April 2008.

[52] A. Vashist, C. A. Kulikowski, and I. Muchnik. Ortholog clustering on a multipartite graph. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, 4(1):17–27, 2007.

[53] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic. Memtracker: Efficient and programmable support for memory access monitoring and debugging. In *HPCA '07*, pages 273–284, 2007.

[54] T. Xie, S. Thummalapenta, D. Lo, and C. Liu. Data mining for software engineering. *IEEE Computer*, Vol. 42(8):35–42, Aug 2009.

[55] Y. Xie and A. Aiken. Context- and path-sensitive memory leak detection. In *ESEC/FSE-13*, pages 115–125, 2005.

[56] G. Xu and A. Rountev. Precise memory leak detection for Java software using container profiling. In *ICSE '08*, pages 151–160, 2008.

[57] X. Yan, H. Cheng, J. Han, and P. S. Yu. Mining significant graph patterns by leap search. In *SIGMOD '08*, pages 433–444, 2008.

[58] X. Yan and J. Han. gSpan: graph-based substructure pattern mining. In *ICDM '02*, pages 721–724, 2002.

[59] X. Yan, X. J. Zhou, and J. Han. Mining closed relational graphs with connectivity constraints. In *KDD '05*, pages 324–333, 2005.

[60] M. J. Zaki and C. Hsiao. Charm: An efficient algorithm for closed itemset mining. In *SDM '02*, pages 457–473, 2002.

[61] Z. Zeng, J. Wang, L. Zhou, and G. Karypis. Out-of-core coherent closed quasi-clique mining from large dense graph databases. *ACM TODS*, 32(2):13, 2007.

Vita

Evan Maxwell is a M.S. student at Virginia Tech in the Department of Computer Science. He is a member of the Softlab research group headed by Dr. Naren Ramakrishnan. He grew up in Vienna, Virginia, and received his B.S. at Virginia Tech in 2008 with a computer science major and mathematics minor. His main academic interests include data mining, bioinformatics, graph theory, and artificial intelligence. He will be joining a bioinformatics Ph.D. program jointly with Boston University and the National Institutes of Health.