Using Substructure Mining to Identify Misbehavior in Network Provenance Graphs

David DeBoer Wenchao Zhou Lisa Singh
Georgetown University
{dpd29, wzhou, singh}@cs.georgetown.edu

ABSTRACT

As distributed systems become more ubiquitous and more complex, the need for efficient, scalable tools to analyze these systems increases. Network provenance graphs offer a rich framework for this task, mapping dependencies between system states and allowing one to explain these states. In this paper, we investigate methods for more efficient substructure mining in the context of network provenance graphs. Specifically, we are interested in identifying frequent substructures that can be used as a feature set for modeling common execution patterns. Knowing these will help network administrators detect nodes in the distributed system that are misbehaving. Therefore, this paper focuses on applying and scaling up substructure mining for network provenance graphs by incorporating a graph database (neo4j) into the substructure mining process and implementing optimizations that improve the efficiency of the substructure mining task. Our results show that the use of the neo4j graph database combined with our algorithmic optimizations greatly improves the run time of our algorithm while not significantly affecting the quality of the substructures returned.

1. INTRODUCTION

The past decade has witnessed the huge success of distributed systems that are deployed for a variety of applications, ranging from cloud computing platforms deployed at data centers, to global-scale peer-to-peer systems. Given their pervasive usage that is closely coupled with daily lives, faults (or misbehavior) of distributed systems can be costly. A large body of research has been dedicated to help system administrators understand and analyze the behavior of distributed systems [6, 9]. Among other proposals, network provenance [18, 19, 20] presents an approach that provides the fundamental functionality required for performing such managerial tasks – the capability to "explain" the existence and changes of system state.

Network provenance captures direct and indirect dependency relationships among system states as a graph, where a system state is modeled as a vertex and dependencies between states are modeled as edges. It reveals the dependencies between system states, and permits system administrators to transitively tie observed faults to their potential causes, and to assess the damage that these faults may have caused to the rest of the system.

While network provenance proves to be useful to determine the origins/causes and effects of specific system state, we envision it offering insights into the overall system execution that can be leveraged to answer questions, such as "*Are there parts of the protocol that are not executing correctly?*", "*Are there system invariants that can be inferred from the execution?*", "*Is the protocol executing efficiently?*" As an initial step towards this vision, we explore a systematic approach to identify potential misbehaviors in a system execution by studying the structural features based on common execution patterns from its corresponding network provenance graph.

Our approach for identifying common execution paths involves discovering interesting (or frequent) substructures in a network provenance graph and using these substructures as the feature set for a basic model to determine if nodes in the network are misbehaving. The intuition of our approach is that network provenance can be viewed as an instantiation of the execution logic (or control flow) of a distributed system; the substructures that yield high compression rates are likely to give insights into the dynamics of the execution logic. For instance, one can use the mined substructures to determine common execution patterns, and using these execution patterns, find nodes in the network which may be executing the protocol improperly. Like in other domains, such as social networkings and scientific workflows, the size of network provenance graphs can be large. As a result, performing analysis on these graphs is becoming more and more difficult. In this paper, we focus on applying and scaling up substructure mining for network provenance graphs by incorporating a graph database into the substructure mining process and implementing algorithmic optimizations that reduce the complexity of substructure mining.

We find it helpful to think about substructure mining as two subproblems: (1) identifying possible substructures that may be interesting, and (2) efficiently finding all instances of these substructures in the graph. Many techniques have been developed for substructure mining, each trying to improve on at least one of these two subproblems. Previous work, such as Subdue[5, 8] and gspan[14], explored ways to efficiently traverse the search space, while using different methods, compression and support respectively, to measure the interestingness of substructures. While these methods reduce the search space considerably, they still do not scale well in terms of graph size because both finding a single substructure is difficult and the number of possible substructures to search is very large. A large body of work investigates efficient indexing strategies for subgraph matching [2, 4, 7, 15, 16, 17, 21]. Our approach is to not build an external index, but to use existing database technologies and algorithmic optimizations to improve the performance. Beginning with a well known substructure mining algorithm, we leverage

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Proceedings of the First International Workshop on Graph Data Management Experience and Systems (GRADES 2013), June 23, 2013, New York, NY, USA.
Copyright 2013 ACM 978-1-4503-2188-4 ...$15.00.

Figure 1: Architectural Overview.

a graph database to improve the performance of the search, and propose simple heuristics that improve the performance in the context of network provenance graphs.

2. SYSTEM DESIGN

Figure 1 presents the architectural overview of our system. It takes as input the network provenance graph generated from the execution of a distributed system, and performs substructure mining using statistics from the network provenance graph and previously discovered interesting substructures. Our system is designed to consider the network provenance graph when the system has reached a stable state. We leave as future work the analysis on network provenance graphs that constantly evolve during the execution of a dynamic system. Because these provenance graphs can be large (100,000s of nodes), we develop an approach to substructure mining that integrates a graph database, taking advantage of its search, indexing, and statistics collection functionality. The identified substructures are then used to provide insights into the execution of the distributed system.

The remainder of this section describes in detail the network provenance model and our approach for substructure mining using a graph database.

2.1 Network Provenance

System Model. For ease of exposition, we adopt a system model that is commonly used in database systems to reason about provenance [20]. In this model, the state of the primary system is represented as *tuples*, and the algorithm which describes how tuples are derived from the system's inputs is represented as *derivation rules* [20][1].

Each node in a distributed system has its own set of tuples, and derivation rules can span multiple nodes. For example, the state of a router r might consist of tuples such as `link(@r,a)` to show that r has a link to a, or `path(@r,b,c)` to show that r knows a path to b with cost c. Here, `link` and `path` are the names of specific relations, and `@r` indicates that the tuple is maintained on r in the `path` relation.

Tuples can either be *base tuples* or *derived tuples*. Base tuples correspond to local inputs that are assumed to be true without derivations, e.g., a list of physical links that is input to a routing protocol. Derived tuples are obtained from other tuples through a derivation rule of the form $\tau @ n$:- $\tau_1 @ n_1 \wedge \tau_2 @ n_2 \wedge \cdots \wedge \tau_k @ n_k$. This is interpreted as a conjunction: tuple τ should be derived on n whenever all τ_i exist on their respective nodes n_i. Having explicit derivation rules makes it very easy to explain the existence of a tuple.

Provenance Model. We can define a *provenance graph* $G = (V, E)$ in which each vertex $v \in V$ corresponds to a tuple and each edge (v_1, v_2) indicates that v_2 was derived from v_1. A complete explanation for the existence of τ would be a subtree that is embedded in G and rooted at τ; the leaves consist of base tuples, which require no further explanation.

Example. To illustrate the provenance model, we consider a simple program MinCost that computes the shortest path between every pair of nodes in a network. The expected execution logic is captured as the following derivation rules:

```
r1 path(@S,D,C) :- link(@S,D,C).
r2 path(@S,D,C1+C2) :- link(@Z,S,C1), bestpath(@Z,D,C2).
r3 bestpath(@S,D,MIN<C>) :- path(@S,D,C).
```

Rule `r1` computes single-hop paths from direct links: "*if there is a link between node `S` and `D` with cost `C`, then there is a path between these two nodes with cost `C`*". Rule `r2` computes a multi-hop path by expanding existing paths: "*if node `Z` has a direct link to its neighbor `S` with cost `C1`, and `Z` has path to `D` with cost `C2`, then there is a path between `S` and `D` with cost `C1+C2`.*" Rule `r3` computes the best paths (with minimal costs) by performing aggregation on the `path` relation.

Consider the example execution on the network topology shown in Figure 2(a), where nodes `a` and `b` faithfully follow the MinCost program, and node `c` is misbehaving as it does not compute multi-hop paths (by applying rule `r2`) and send them to its neighbors. There could be multiple reasons why node `c` misbehaves, for instance, node `c` could be selfish and not willing to carry traffic for other nodes (by not including itself as an intermediate node in a path), or `c` could simply encounter software bugs.

Figure 2(c) shows the provenance for the execution of the MinCost program. It encodes the dependency relationship between system states distributed across the nodes. For instance, the best path ($a \rightarrow b$) between node `a` and `b` (denoted `bestpath(@a,b,4))`) is derived from the direct link between these two nodes (denoted as `link(@a,b,4)`) by applying derivation rule `r1` and rule `r3`, and it is then combined with the direct link between node `a` and `c` (denoted as `link(@a,c,1)`) to generate a two-hop path $c \rightarrow a \rightarrow b$ (denoted as `path(@c,b,5)`) that is sent to node `c`.

Figure 2(b) shows two frequent substructures of the provenance graph. Substructure 1 encodes a common pattern of generating one-hop best paths, and Substructure 2 encodes how multi-hop paths (not necessarily with minimal cost) are generated. Note that each of these substructures is a combination of multiple derivation rules, and gives insights on how different parts of the execution logic interact with each other. While Substructure 1 manifests multiple times in the provenance graph for each of the three nodes, no instance of Substructure 2 appear on the provenance graph for node `c`. This corresponds to the misbehavior of node `c` since `c` does not perform the computation of multi-hop paths. A network administrator can use the frequent substructures to focus in on which nodes are missing *expected behavior* and which nodes have unusual, non-frequent behavior in an effort to identify potential misbehavior.

2.2 Substructure Mining

Let $G = (V, E)$ represent a graph, where each vertex in V and each edge in E has a semantic label that is not unique. Let H represent a substructure in G, where $V(H) \subset V(G)$, $E(H) \subset E(G)$ and there is a function $f : V(H) \Rightarrow V(G)$ st $\forall v_i, v_j \in V(H), (v_i v_j) \in E(H), (f(v_i), f(v_j)) \in E(G)$ maps labels to each vertex and edge in G. A substructure H can appear multiple

[1]Few practical systems are explicitly built in terms of tuples and derivation rules. Prior work [18] has described general techniques for extracting tuples and derivation rules from existing systems.

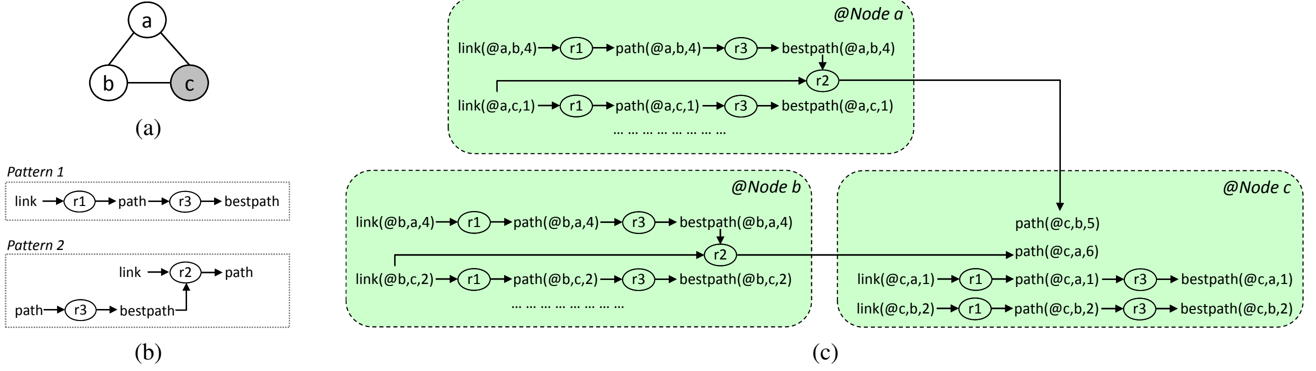

Figure 2: An example provenance graph for an execution of the MinCost program. Figure (a) shows the example network topology, where the misbehaving node is shaded; Figure (b) presents two frequent patterns in the provenance graph, which can be used as the features of the execution of the MinCost program; Figure (c) denotes a (partial) provenance graph of the program execution. Note the differences of the structure features between the provenance graphs for node c and the other two nodes (e.g., the provenance graph for node c does not contain instances of Substructure 2).

times in G using different functions which correspond to different vertices and edges in G and will be referred to as different instances of H in G. H' is an *extension* of substructure H that contains $V(H)$ and $E(H)$ along with one or more additional vertices and edges not in H.

Problem. Given a graph $G = (V, E)$, identify the set of substructures $H = \{H_1, H_2, ...H_n\}$ that are frequent and lead to the highest compression rate. Here, a frequent substructure is one that appears often in G. If every instance of H_i in G is reduced to a single vertex, we define the compression rate as $\frac{\phi(G)}{\phi(H_i)+\phi(G|H_i)}$, where $\phi(G)$ is the description length of G, H_i is the substructure, and $G|H_i$ is G after being compressed based on H_i (see Figure 3 for detailed equation).

Approach. We use a graph database to store the full graph and the label index to take advantage of the database's optimizations.For large graphs, we surmise that the database will improve the performance over a memory-based implementation.

Algorithm 1 shows the high-level algorithm for substructure mining that incorporates the use of a graph database. Given a graph G, the algorithm begins by identifying a substructure to search for (identify_initial_candidate_substructures()). The database is then queried for those substructures and all the matching substructures are returned (find_matches()). If the substructures are frequent and lead to high compression ($compress > 1$), they are maintained in H and in the temporary *possible candidate set* (C'). The top $candidate_set_size$ substructure matches based on compression (get_top_matches()) are maintained. These substructures are extended in all directions to generate more substructures to search for (extend_substructure()). This process continues until the maximum number of expansions specified by the user is reached. Finally, the substructures that are most frequent and lead to the highest compression are returned as the result set (H).

Note that different stopping criteria can be used for substructure mining, such as criteria defined on the maximum size of the substructure, the maximum number of iterations, or the expected gain if the search is continued. One of the challenges of substructure mining is that each iteration involves searching the graph for a candidate subgraph, which corresponds to the subgraph isomorphism problem that has been shown to be NP-Complete [3].

A number of different algorithms have been proposed that are variants of the basic algorithm described in Algorithm 1. While any of these algorithms are a reasonable starting point, we choose to use Subdue [5] as our base algorithm because it follows the minimum description length (MDL) principle, which we use to theoretically ground our work. The MDL principle states that "the best theory to describe a set of data is that theory which minimizes the description length of the entire data set" [5]. Using the MDL principle, the substructures that are found can be ranked based on how well they compress the overall graph. In addition, Subdue allows us to successfully limit the complexity of both aspects of substructure mining — the search space of substructures and the complexity of finding substructure instances within a graph.

Algorithm 1 High Level Substructure Mining

1: **Input:** G, $candidate_set_size$, $max_nbr_expansions$
2: **Output:** H
3:
4: $H = \emptyset$, $C = \emptyset$, $nbr_expansions = 0$
5:
6: C = identify_initial_candidate_substructures(G)
7: **while** $nbr_expansions < max_nbr_expansions$ **do**
8: $\quad C' = \emptyset$
9: $\quad$ **for all** c in C **do**
10: $\quad\quad instanceCount$ = find_matches(G, c)
11: $\quad\quad$ **if** $compress > 1$ **then**
12: $\quad\quad\quad H = H \cup c$
13: $\quad\quad\quad C' = C' \cup c$
14: $\quad C'$ = get_top_matches(C', candidate_set_size)
15: $\quad C = \emptyset$
16: $\quad$ **for all** c' in C' **do**
17: $\quad\quad C = C \cup$ extend_substructure(c')
18: $\quad\quad nbr_expansions$ ++
19: **return** H

$$
\begin{aligned}
compress &= \frac{\phi(G)}{\phi(H) + \phi(G|H)} \\
\phi(G) &= |V(G)| + |E(G)| \\
\phi(H) &= |V(H)| + |E(H)| \\
\phi(G|H) &= (|V(G) - |V(H)| * instanceCnt + instanceCnt) \\
&\quad + (|E(G) - |E(H)| * instanceCnt)
\end{aligned}
$$

Figure 3: Calculation of compression rate [1]. Here, $instanctCnt$ is the number of times substructure H appears in G and ϕ represents the description length

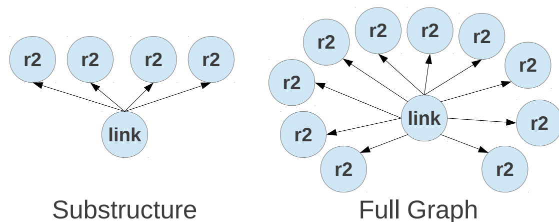

Figure 4: There are $\binom{10}{4}$ instances of the substructure query graph in the full graph.

3. OPTIMIZATIONS

In this section we present a number of optimizations for reducing the cost of each of the subgraph mining subproblems: identifying candidate substructures that may be interesting and efficiently finding all instances of H in G. While these optimizations do improve the efficiency of the basic graph mining algorithm, they can also lead to missing interesting substructures. However, we show in the evaluation section that for provenance graphs, our accuracy remains high.

3.1 Finding Interesting Substructures

As mentioned in the previous section, to find high compression substructures, we iteratively expand a fixed number of high compression substructures, i.e. we perform a beam search. Two heuristics that improve our basic search follow.

Duplicate Substructure Reduction (DUP-REDUCE). One problem with our basic algorithm is that during the substructure expansion, it is possible to generate duplicate substructures during one iteration of the expansion, as well as across multiple iterations. The mining efficiency can be significantly improved by reducing (or eliminating) the duplicates. Therefore, the subgraph duplication reduction heuristic augments the beam search with the canonical labeling of substructures and the closed subgraph concept used in gspan and closegraph [14]. Instead of expanding all possible paths, this heuristic only expands substructures along a specific path of a spanning tree of the substructure. Also, if the instance count remains the same when a vertex is added to the substructure, the substructure is considered "closed" because any other expansion on that substructure will be the same as expanding the new larger substructure. However, because this method uses instance count as its metric rather than description length, it is possible that the search may miss substructures that have higher compression. Our results show that this is not the case for our network provenance graphs.

Outward Expansion (EXPAND-OUT). This heuristic focuses on limiting the types of edges used during substructure expansion. Because we are working with directed provenance graphs, and a provenance graph models a process of data dependencies, expanding substructures using only outgoing edges is meaningful in the context of rule execution. While this limits the types of substructures that will be found, it answers the question "Given this piece of information, what execution patterns can be derived from it?". Therefore, this optimization expands substructures using only outgoing edges, thereby reducing the number of substructures that must be compared to the original graph.

3.2 Query Graph Matching

While the number of substructures that are considered improves the performance of the basic substructure mining algorithm, the query graph matching subproblem is the more costly step. Therefore, improving the efficiency of this subproblem will have a larger impact on the overall runtime.

Figure 4 shows an example of a small graph (right) and a corresponding substructure (left). For this small example, there are $\binom{10}{4}$ $= 5040$ possible instances of this substructure. While this type of structure may be uncommon for social networks, it is not unusual for network provenance graphs to have a large number of overlapping matches within a small graph, e.g. a single vertex can have 50 or more edges to vertices with the same label. To help reduce the impact of overlaps, we use a set of start nodes to focus our search. To get this set, a vertex in the candidate substructure is chosen, and all nodes in the full graph with the same label as that vertex are selected as the set of start nodes. Each start node is a possible starting point for a substructure instance. Once a single match is found for a start node, we move on to the next start node. We propose two different heuristics which take advantage of this set of start nodes:

Infrequent Start Vertex (LESS-FREQ). Deciding which vertex of the substructure to start with will impact the size of the set of start nodes. For example, if a frequent vertex is selected as the start vertex, then there is a higher number of candidate instances in G that the substructure needs to be compared against. On the other hand, if the vertex that starts the search is infrequent, then the number of candidates to compare against is lower. Therefore, this optimization uses the vertex with a label that appears least frequently in G as the start vertex for subgraph matching.

Start Vertex Reuse (REUSE). All substructures are extended versions of substructures that have already been examined. This heuristic uses knowledge about the matched instances of H_i as a starting point when searching for instances of query substructure H_i'. The intuition is that since H_i' is extended from and therefore contains H_i, starting with the instances of H_i as candidate matches will focus the search for instances of H_i' to the candidates that are most likely to be a match. Since maintaining all the instances of H_i can be expensive in terms of memory usage, we only reuse the start vertex of the instances of H_i. In other words, we *reuse* the start vertices from H_i during our match search for instances of H_i'.

4. PRELIMINARY EVALUATION

4.1 Prototype Implementation

We implemented two optimized versions of our algorithm, an in-memory version (MEM-OPTMIZED) and a graph database version (DB-OPTIMIZED) using the neo4j graph database. We represented our graphs and substructures as lists of vertices, where each vertex had a list of adjacent edges associated with it. We built an external index that mapped a vertex label to the set of vertices with that label. (For the neo4j implementation, we also used its auto-indexing feature). Traversals for substructure matching were performed using the vertex index for the initial lookup, and then using the edge lists for each vertex to traverse the graph for matches.

Since the neo4j query language, Cypher, did not support returning only a single instance of a substructure without finding all the other instances (this is important for finding non-overlapping substructures), we used a depreciated set of neo4j classes for substructure matching and modified them to take advantage of the heuristics mentioned in Section 3.

Because we were interested in identifying non-overlapping instances of a substructure, we maintained a hash table of used vertices in memory. [2] We then explicitly checked each matching instance returned to see if it contained used vertices, removing those that did. We also adjusted our substructure matching to improve the runtime of matching. Our approach was to partition a large

[2]We attempted to keep track of this efficiently within neo4j, but we were unable to find an efficient option.

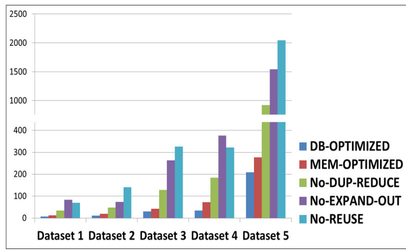

Figure 5: Completion Times (s).

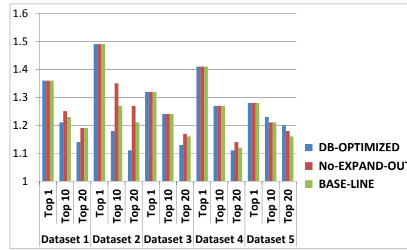

Figure 6: Compression.

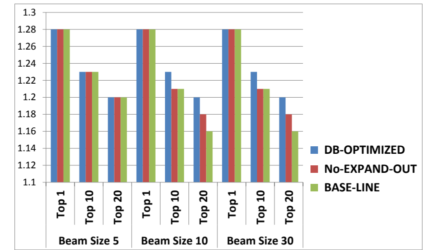

Figure 7: Sensitivity to Beam Size.

Dataset	ASN	Nodes	Links	$\|V(G)\|$	$\|E(G)\|$
1	1221	108	306	16,227	28,090
2	1755	87	322	23,015	40,725
3	3257	161	656	52,848	94,568
4	6461	141	748	73,316	134,072
5	1239	315	1,944	317,066	592,038

Table 1: RocketFuel Topologies

substructure into small substructure pieces (or *patterns*), similar in principle to the idea used in [17]. Each vertex v adjacent to the start vertex was considered to belong to a separate pattern, and all other vertices, reachable from v without traversing the start vertex, were part of that pattern (or path). We then performed substructure matching for each of the patterns individually. This improved the efficiency because it was faster to search for smaller substructures and easier to identify overlaps with previous patterns.

4.2 Performance Evaluation

We performed experimental evaluations to study the performance of substructure mining on network provenance graphs and the effectiveness of our proposed optimizations.

Experimental Setup. Our experiments were performed on a machine with two 2.13 GHz Intel Pentium P6200 and 4 GB of memory. We derive our test provenance graphs from executions of the MinCost protocol, on several inferred intra-domain topologies reported in the RocketFuel project [11]. The topologies and the generated provenance graphs are summarized in Table 1. All of our evaluations were performed 11 times, with the first result being discarded to account for warm up time. Our results show the average of the 10 remaining runs.

Methods. MEM-OPTIMIZED and DB-OPTIMIZED are the in-memory and neo4j-based implementations with all the optimizations presented in Section 3. We compare these two methods with neo4j-based implementation that did not apply any of the optimizations (BASELINE). To validate the effectiveness of each individual optimization, we additionally evaluate three methods: For "No-REUSE", we maintain all the optimizations except the "Start Vertex Reuse" optimization. For "No-EXPAND-OUT", we maintain all the optimizations except the "Outward Expansion" heuristic. And for "No-DUP-REDUCE", we do not use the "Substructure Duplication Reduction" heuristic.

Evaluation Results. We now present our evaluation results in terms of the completion time and compression rate to study the trade offs between efficiency and accuracy when different optimizations were applied. We also studied the sensitivity of the results to the beam size ($nbr_expansions$) during each iteration.

- **Completion time.** The evaluation results are presented in Figure 5. All of our evaluations perform 100 substructure expansions of the best substructures. Our results clearly show that not only do our optimized methods perform best, but also the absence of even a single heuristic affects the runtime greatly. The "Start Vertex Reuse" heuristic is the biggest time saver for most graphs, and "Outward Expansion" also makes a large contribution. Finally, it is notable that the optimized implementation using the database outperforms the in-memory version.

- **Compression rate.** Figure 6 summarizes compression rates for the best substructure (top 1), the 10th best (top 10), and the 20th best (top 20) across the 5 data sets. We did not observe any difference in compression rates for No-REUSE and No-DUP-REDUCE, and omitted them from the figure to reduce complexity. From this figure, we can see that for our graphs and settings, the top 1 is always the same across methods. No-EXPAND-OUT and BASELINE perform better at 10 and 20 for some graphs and worse for others.

  We observed that DB-OPTIMIZED performed worse as it missed one type of substructure which was found by No-EXPAND-OUT and BASELINE. This is understandable, as the "Outward Expansion" heuristic restricts the types of candidates substructures. DB-OPTIMIZED performed better in data set 5, because the No-EXPAND-OUT and BASELINE methods did not reach the same size of substructures as DB-OPTIMIZED. This occurred because those methods considered smaller substructures and reached the 100 expansions limit (beam size) before they reached the larger substructures. This result emphasizes the trade off between compression rate, efficiency, and the top substructures returned.

- **Sensitivity to beam size.** In Figure 7 we show the change in compression when varying the beam size from 5 to 30. Our evaluation was performed using the largest graph (Dataset 5) and with the number of expansions set to 100. We observed that varying the beam size generally did not have a large effect on the final compression results. Only changing from beam size 5 to beam size 10 had an effect on the compression, and only for the BASELINE and No-EXPAND-OUT methods. Comparing the returned substructures, we noticed that the BASELINE and No-EXPAND-OUT did not reach large substructures with beam sizes 10 and 30. This case was mentioned previously - these methods analyzed smaller substructures and reached the 100 expansion limit before analyzing any larger substructures. A smaller beam size allowed us to reach larger substructures quicker, searching fewer substructures, while a larger beam allows us to consider more substructures during each iteration.

5. DISCUSSION

Our results provide initial evidence that it is possible to find high-compression substructures in network provenance graphs, even as the graphs become very large. Being able to leverage these high-compression substructures in analyzing the execution of a network protocol should enable us to identify malicious behavior for nodes

in a network. Our algorithm allows us to search many possible substructures within the full provenance graphs to find those that are most indicative of normal behavior. Also, once a common set of substructures are found, this methodology will further allow us to quickly query a provenance graph for a specific network node to determine if it is exhibiting malicious behavior.

We are also encouraged by our results using the graph database neo4j. The run times for the large graphs we tested are acceptable for our application. We are surprised that the database version consistently outperforms our memory algorithm. Our initial analysis has not shown conclusively why this is the case, or what aspects of neo4j gives it an advantage in our implementation. We leave a more detailed analysis for future work. To get this performance, we explicitly used many parts of the neo4j API, including some depricated classes and methods. It seems that at least some of what we have implemented could be included as part of a graph database to facilitate frequent substructure mining. Some additional database functions that would support this include:

- Returning only non-overlapping instances of substructures.
- Returning a single instance of a substructure.
- Having a caching mechanism that stores information about intermediate substructure instances to aid expanded substructure matching.

For our application of frequent substructure mining, and possibly substructure mining in other domains, these improvements would make this type of analysis with a graph database much easier. While frequent substructure mining is only a single application for graph databases, it is an important one, and improving how graph databases handle substructure mining would increase their overall usefulness.

6. RELATED WORK

Given the increasing size and complexity of graphs, taking advantage of database technologies and methodologies for frequent substructure mining is an important direction of research. A number of memory based substructure mining algorithms have been proposed in the literature [5, 8, 14]. Gspan and closegraph [14] use the apriori-like idea of minimum support to bound the search space and identify frequent subgraphs. Subdue [5, 8] uses the principle of MDL to identify substructures that compress the graph. While we utilize components of each of these algorithms in our work, we augment them to 1) identify only non-overlapping frequent substructures; and 2) incorporate the use of a graph database. Padmanabhan and Chakravarthy propose an implementation of Subdue in a relational database [10]. However, because new tables are created for each interesting substructure, the number of tables increases rapidly for data sets with a large number of frequent substructures. To the best of our knowlede, we are the first to investigate this problem using an existing graph database.

One of the subproblems in frequent substructure mining is subgraph matching. Approaches for subgraph matching can be divided into two categories, those that propose new indexing strategies [2, 4, 7, 15, 16, 17, 21] and those that do not build an index structure [12, 13]. We focus on those methods that do not build an external index since our focus is on in-memory solutions and solutions using a database's existing infrastructure. Tian et. al [13] propose SAGA, a subgraph matching tool for biological graphs. They focus on approximate matches using a graph distance metric to measure similarity between graphs. Our focus differs since we are interested in exact matches using larger graphs stored in a graph database. In a tangential approach to our work, Sun et. al [12] handle very large graphs using a distributed memory store, partitioning the graph and the query, and parallelizing the query processing.

7. CONCLUSION

This paper presents a method for quickly finding high compression substructures within provenance graphs with the explicit purpose of identifying common execution patterns for network protocols and using these patterns to identify malicious nodes in a network. We demonstrate that it is possible to use some simple heuristics to speed up the substructure mining process, and that using a graph database like neo4j can lead to improvements in the performance of the substructure mining algorithm. Future work will focus on testing these methods against provenance graphs created by different protocols and with different malicious behavior. Considering other graph databases, particularly ones that support hypernodes and hyperedges, is also an important direction. Another possible extension involves studying the mining of provenance graphs which are dynamically changing in a live environment rather than in a static offline state. Frequent substructures are just one facet of the rich set of features which network provenance graphs offer. Pairing these graphs with the expanding functionality and analytical promise of graph databases should lead to interesting research opportunities.

8. REFERENCES

[1] R. Balachandran, S. Padmanabhan, and S. Chakravarthy. Enhanced db-subdue: Supporting subtle aspects of graph mining using a relational approach. In *Advances in Knowledge Discovery and Data Mining*. 2006.

[2] J. Cheng, J. Yu, B. Ding, P. Yu, and H. Wang. Fast graph pattern matching. In *Proc. ICDE*, pages 913–922, 2008.

[3] S. A. Cook. The complexity of theorem-proving procedures. In *Proc. STOC*, 1971.

[4] H. He and A. K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *Proc. SIGMOD*, pages 405–418, 2008.

[5] L. B. Holder, D. J. Cook, S. Djoko, et al. Substructure discovery in the subdue system. In *Proc. of the AAAI Workshop on Knowledge Discovery in Databases*, 1994.

[6] G. Jiang, H. Chen, and K. Yoshihira. Efficient and scalable algorithms for inferring likely invariants in distributed systems. *IEEE TKDE*, 19(11), 2007.

[7] H. Jiang, H. Wang, P. Yu, and S. Zhou. Gstring: A novel approach for efficient search in graph databases. In *Proc. ICDE*, pages 566–575, 2007.

[8] N. S. Ketkar, L. B. Holder, and D. J. Cook. Subdue: compression-based frequent pattern discovery in graph data. In *Proc. OSDM*, 2005.

[9] J.-G. Lou, Q. Fu, S. Yang, Y. Xu, and J. Li. Mining invariants from console logs for system problem detection. In *Proc. of ATC*, 2010.

[10] S. Padmanabhan and S. Chakravarthy. Hdb-subdue: A scalable approach to graph mining. *Data Warehousing and Knowledge Discovery*, 2009.

[11] N. Spring, R. Mahajan, and D. Wetherall. Measuring isp topologies with rocketfuel. *ACM SIGCOMM CCR*, 32(4), 2002.

[12] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *Proc. VLDB Endow.*, 5(9):788–799, May 2012.

[13] Y. Tian, R. C. Mceachin, C. Santos, D. J. States, and J. M. Patel. Saga: a subgraph matching tool for biological graphs. *Bioinformatics*, 23(2):232–239, Jan. 2007.

[14] X. Yan and J. Han. Closegraph: mining closed frequent graph patterns. In *Proc. SIGKDD*, 2003.

[15] Y. Yuan, G. Wang, L. Chen, and H. Wang. Efficient subgraph similarity search on large probabilistic graph databases. *Proc. VLDB Endow.*, 5(9):800–811, May 2012.

[16] S. Zhang, S. Li, and J. Yang. Gaddi: distance index based subgraph matching in biological networks. In *Proc. EDBT*, pages 192–203, 2009.

[17] P. Zhao and J. Han. On graph query optimization in large networks. *Proc. VLDB Endow.*, 3(1-2):340–351, Sept. 2010.

[18] W. Zhou, Q. Fei, A. Narayan, A. Haeberlen, B. T. Loo, and M. Sherr. Secure network provenance. In *Proc. SOSP*, 2011.

[19] W. Zhou, S. Mapara, Y. Ren, A. Haeberlen, Z. Ives, B. T. Loo, and M. Sherr. Distributed time-aware provenance. In *Proc. VLDB*, 2013.

[20] W. Zhou, M. Sherr, T. Tao, X. Li, B. T. Loo, and Y. Mao. Efficient querying and maintenance of network provenance at Internet-scale. In *Proc. SIGMOD*, 2010.

[21] L. Zou, L. Chen, and M. T. Özsu. Distancejoin: Pattern match query in a large graph database. *PVLDB*, 2(1):886–897, 2009.