Discovering Frequent Structures using Summaries

Shayan Ghazizadeh
Department of Computer Science
University of Maryland, College Park
shayan@cs.umd.edu

Sudarshan Chawathe
Institute for Advanced Computer Studies
University of Maryland, College Park
chaw@cs.umd.edu

Abstract

We study the problem of finding frequent structures in semistructured data (represented as a directed labeled graph). Frequent structures are graphs that are isomorphic to a large number of subgraphs in the data graph. Frequent structures form building blocks for visual exploration and data mining of semistructured data. We overcome the inherent computational complexity of the problem by using a summary data structure to prune the search space and to provide interactive feedback. We present an experimental study of our methods operating on real datasets. The implementation of our methods (which is freely available) is capable of operating on datasets that are two to three orders of magnitude larger than those described in prior work.

1 Introduction

Technological factors such as the falling prices and increasing capacities of storage media have made it practical for organizations to store extremely large amounts of data generated by their operations. Indeed, apart from secrecy and legal protection, there is very little motivation for permanently deleting any data since the cost of storing it is often negligible compared to its potential impact on decision making and future operations. However, while storing data is easy, making effective use of it is very difficult because the amount of data is several orders of magnitude larger than what a human expert can analyze. As a result, there is a pressing need for *data mining*, a term we use to denote the semi-automatic extraction of interesting patterns from large amounts of data. Although there has been considerable recent work on data mining in both research and product communities, most of it has focused on relational or otherwise well structured data (such as time series or high-dimensional data). On the other hand, it is well recognized that a significant portion of the information vital to an enterprise is not well structured, but is either unstructured or semistructured. We use the term *semistructured data* to mean data whose structure is irregular, incomplete, and dynamic. For example, a collection of memos or legal documents has significant structure (e.g., sender's name, subject, dates, outcomes, etc.); however this structure is not as regular or reliable as that found in traditional (well structured) databases. In this paper, we address the problem of mining such semistructured data.

In many data mining tasks, an important (and frequently most time-consuming) task is the discovery and enumeration of *frequently occurring patterns*, which are informally sets of related data items that occur frequently enough to be of potential interest for a detailed data analysis. The precise interpretation of this term depends on the data model, dataset, and application. Perhaps the best studied framework for data mining uses association rules to describe interesting relationships between sets of data items [AIS93]. In this framework, which is typically applied to market basket data (from checkout registers, indicating items purchased together), the critical operation is determining *frequent itemsets*, which are defined as sets of items that are purchased together often enough to pass a given threshold (called the support). For time series data, an analogous concept is a subsequence of the given series that occurs frequently. This paper defines an analogous concept, called *frequent structures* for semistructured data (represented as a labeled directed graph) and presents efficient methods for computing frequent structures in large datasets.

Data mining is an iterative process in which a human expert refines the parameters of a data mining system based on intermediate results presented by the mining system. It is unreasonable to expect an expert to select the proper values for mining parameters a priori because such selection requires a detailed knowledge of the data, which is what the mining system is expected to enable. While frequent and meaningful feedback is important for any data mining system, it is of particular importance when the data is semistructured because, in addition to the data-dependent relationships being unknown a priori, even the schema is not known (and not fixed). Therefore, rapid and frequent feedback to a human expert is a very important requirement for any system that is designed to mine semistructured data. Prior work (discussed in Section 4) on mining such data often falls short on this requirement.

The main idea behind our method, which is called *SEuS* (for Structure Extraction using Summaries), is the following three-phase process: In the first phase (*summarization*), we preprocess the given dataset to produce a concise summary. This summary is an abstraction of the underlying graph data and it indicates the types of relationships between nodes identified using their labels. Our summary is thus similar to data guides and other (approximate) typing mechanisms for semistructured data [GW97, BDFS97, NUWC97, NAM97]. As has been noted in such work (and confirmed by our experiments in Section 3) such summaries are typically dramatically smaller than the underlying database. In the second phase (*candidate generation*), our method interacts with a human expert to iteratively search for frequent structures and refine the support threshold parameter. Since the search uses only the summary, which typically fits in main memory, it can be performed very rapidly (interactive response times) without any additional disk accesses. Although the results in this phase are approximate (a supper set of final results), they are accurate enough to permit uninteresting structures to be filtered out. (We discuss the nature of the approximation in Section 2.5.) When the expert has filtered potential structures using the approximate results of the search phase, an accurate count of the number of occurrences of each potential structure is produced by the third phase (*counting*). As we shall see in Section 3, this phase accounts for the majority of the time spent on the mining process.

Users are often willing to sacrifice quality for a faster response. For example, during the preliminary exploration of a dataset, one might prefer to get a quick and approximate insight into the data and base further exploration decisions on this insight. In order to address this need, we introduce an approximate version of our method, called L-SEuS. This method only returns the top-n frequent structures rather than all frequent structures.

The methods in this paper have three significant advantages over prior work: First, they operate efficiently on datasets that are two to three orders of magnitude larger than those handled by prior work of which we are aware. Second, even for large datasets, our methods provide approximate results very quickly, enabling their use in an interactive exploratory data analysis. Third, for applications and scenarios that are interested in only the frequent structures, but not necessarily their exact frequencies, the most expensive counting phase can be completely skipped, resulting in great performance benefits.

In order to evaluate our ideas, we have implemented our method in a data mining system for (semi)structured data (also called SEuS). In addition to serving as a testbed for our experimental study (Section 3), the system is useful in its own right as a tool for exploring (semi)structured data. We have found it to discover intuitively meaningful structures when applied to datasets from several domains. Our implementation of SEuS uses the Java 2 (J2SE) programming environment and is freely available at `http://www.cs.umd.edu/projects/seus/` under the terms of the GNU GPL license.

The rest of this paper is organized as follows: In Section 2, we define the structure discovery problem formally and present our three-phase solution called SEuS. Sections 2.1, 2.2, and 2.3 describe the summarization, candidate generation, and counting phases. Section 2.4 presents our approximate method called L-SEuS. In Section 2.5, we discuss the quality of solutions produced by our complete and approximate methods. Section 3 summarizes the results of our detailed experimental study. Related work is discussed in Section 4 and we conclude in Section 5.

2 Structure Discovery

SEuS represents semistructured data as a labeled directed graph. In this representation, objects are mapped to vertices and relations between these objects are modeled by edges. A *structure* is defined to be a connected graph that is isomorphic to at least one subgraph of the database. Figure 1 illustrates the graph representation of a small XML database. Any subgraph of the input database that is isomorphic to a structure is called an *instance* of that structure. The number of instances of a structure is called the structure's *support*. (We allow the instances to overlap.) For the data graph in Figure 1, a structure and its three instances are shown in Figure 2. We say a structure is *T-frequent* if it has a support higher than a given *threshold* T. **Problem statement (frequent structure discovery):** Given the graph representation of a database and a threshold T, find the set of *T-frequent structures*.

A naive approach for finding frequent structures consists of enumerating all subgraphs, partitioning this set of subgraphs into classes based on graph isomorphism, and returning a representative from the classes with cardinality greater than the support threshold. Unfortunately, the number of subgraphs of a graph database is exponential in the size of the graph. Further, the naive approach tests each pair of these subgraphs for isomorphism in worst case.

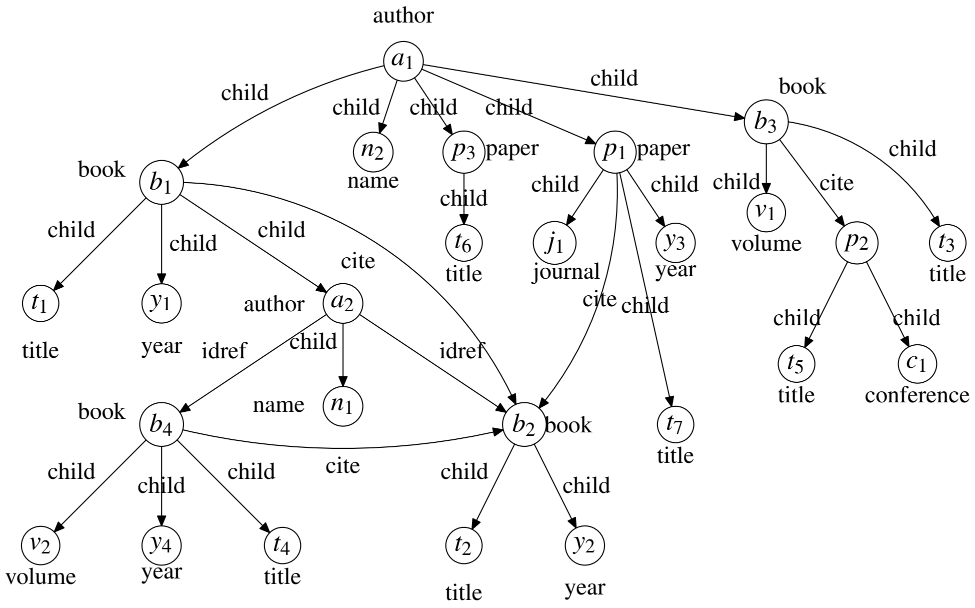

Figure 1: Example input graph

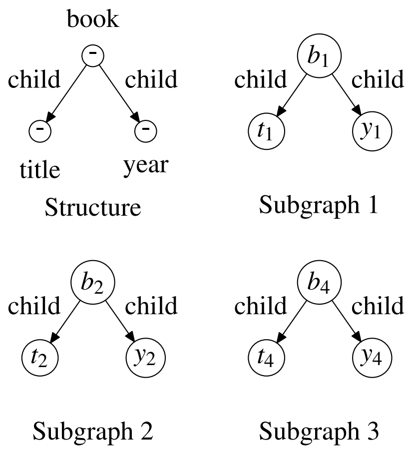

Figure 2: A structure and its three instances

Although graph isomorphism is not known to be NP-hard (or in P) [For96], it is a difficult problem and an approach relying on an exponential number of isomorphism tests is unlikely to be practical for large databases.

Although we do not have a proof for the hardness of the frequent structure discovery problem, the closely related problem of finding the support of the most frequent k-vertex structure is NP-hard. The hardness result follows by reduction from the NP-hard problem of deciding whether there exists a k-vertex clique in a given n-vertex graph G. For this purpose we construct a new graph H consisting of a copy of G and a disjoint copy of K_{n+1}, the complete graph on $n+1$ vertices. Clearly, the most frequent k-vertex subgraph in H is a clique of k vertices. If G has no k-sized cliques, then k-sized cliques occur exactly $t = C_k^{n+1}$ times in H. If G does have a k-sized clique, then k-sized cliques occur at least $t+1$ times. So, if we can count the support of the most frequent k-vertex subgraph in H, we can decide whether G has a k-clique.

Given the above, practical systems must use some way to avoid examining all the possible subgraphs and must calculate the support of structures without partitioning the set of all possible subgraphs. Instead of enumerating all of the subgraphs in the beginning, we can use a level-by-level expansion of subgraphs similar to the k-itemset approach adopted in Apriori [AS94] for market basket data. We start from subgraphs of size one (single vertex) and try to expand them by adding more vertices and edges. A subgraph is not expanded anymore as soon as we can reason that its support will fall under the threshold based on *downward closure property*: A structure has a support higher than a threshold if all of its subgraphs also have a support higher than the threshold. AGM [IWM00] and FSG [KK01] are two recent systems that adopt this strategy to find all structures with a support higher than a given threshold. SUBDUE [CH00] is a greedy method that finds frequent structures in a graph database using the minimum description length (MDL) principle. This method is not complete in the sense that it may not obtain all frequent structures. SUBDUE also uses the level-by-level expansion. However, instead of expanding all subgraphs at each iteration, SUBDUE only expands the subgraphs belonging to the isomorphism class with the highest score based on the MDL principle. To bound the running time, SUBDUE uses a beam search that is computationally constrained. (For more details see Section 4.)

The results reported in [IWM00, KK01, CH00], as well as our experiments, suggest that these methods do not scale to very large databases. For a graph with 9000 vertices (which is much smaller than the datasets that interest us, consisting of several million vertices), [KK01] reports that AGM needs about 8 days and FSG will take 600 seconds for thresholds around 10%. Although SUBDUE only takes 80 seconds to process the same dataset, our experiments show that it does not scale easily to larger datasets. For example, it takes SUBDUE longer than 24 hours to mine a 50 megabyte dataset. (This experiment was run on a PC-class machine using the serial implementation of SUBDUE. For more details see Section 3.) The main factor hurting performance of these methods is the need to go through the database to determine the support of each structure. Although the number of structures for which the support has to be calculated has decreased significantly compared to the naive approach (due to the use of downward closure properties or MDL heuristic), the calculation of the support of the remaining structures is still expensive. Further, all of these systems operate in a batch mode: After providing the input database, a user has to wait for the structure discovery process to terminate before any output is produced. There are no intermediate (partial or approximate) results, making exploratory data analysis difficult. This batch mode operation can cause major problems, especially when the user does not have enough domain knowledge to guess proper values for mining parameters (e.g., support threshold).

In order to operate efficiently, SEuS uses *data summaries* instead of the database itself. Summaries provide a concise representation of a database at the expense of some accuracy. This representation allows our system to approximate the support of a structure without scanning the database. We also use the level-by-level expansion method to discover frequent structures. SEuS has three major phases: The first phase (*summarization*) is responsible for creating the data summary and is described in Section 2.1. In the second phase (*candidate generation*), SEuS finds all structures that have an estimated support above the given threshold; it is described in Section 2.2. The second phase reports such candidate structures to the user, and this early feedback is useful for exploratory work. The exact support of structures is determined in the third phase (*counting*), described in Section 2.3.

2.1 Summarization

We use a data summary to estimate the support of a structure (i.e., the number of subgraphs in the database that are isomorphic to the structure). Our summary is similar in spirit to representative objects, graph schemas, and *DataGuides* [NUWC97, BDFS97, GW97]. The summary is a graph with the following characteristics. For each distinct vertex label l in the original graph G, the summary graph S has an l-labeled vertex. For each m-labeled edge (v_1, v_2) in the original graph there is an m-labeled edge (l_1, l_2) in S, where l_1 and l_2 are the labels of v_1 and v_2, respectively. The summary S also associates a counter with each vertex (and edge) indicating the number of vertices (respectively, edges) in the original graph that it represents. For example, Figure 3 depicts the summary generated for the input graph of Figure 1.

Since all vertices in the database with the same label map to one vertex in the summary, the summary is typically much smaller than the original graph. For example, the graph of Figure 1 has four vertices labeled *book*, while the summary has only one vertex representing these four vertices. In this simple example, the summary is only slightly smaller than the original data. However, as noted in [GW97], many common datasets are characterized by a high degree of structural replication, giving much greater space savings. (For details, see Table 2 in the Appendix.) These space savings come at the cost of reduced accuracy of representation. In particular this summary tells us the labels on possible edges to and from the vertices labeled *paper*, although they may not all be incident on the same vertex in the original graph. (For example, *journal* and *conference* vertices never connect to the same *paper* vertex, but the summary does not contain this information.)

We can partly overcome this problem by creating a richer summary. Instead of storing only the set of edges leaving a vertex label and their frequencies, we can create a *counting lattice* (similar to the one used in [NAM97]), $L(v)$ for each vertex v. For every distinct set of edges leaving v, we create a node in $L(v)$ and store the frequency of this set

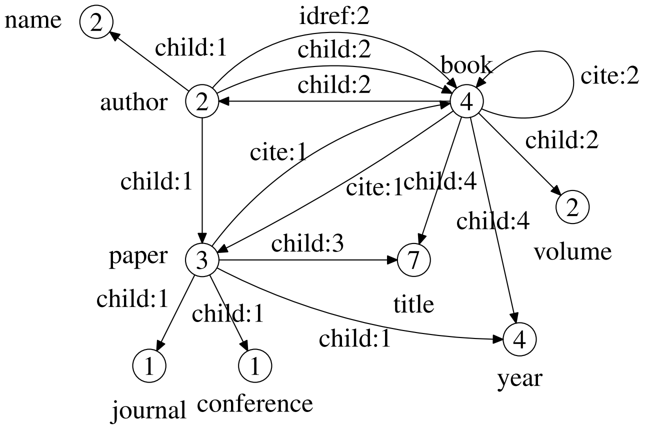

Figure 3: Summary graph

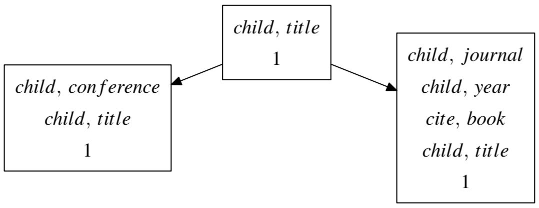

Figure 4: Counting Lattice for *paper* vertex

of outgoing edges. For example, consider the vertex label *paper* in Figure 1. The counting lattice for this vertex is depicted in Figure 4. In the input graph, there are three different types of *paper* vertices with respect to their outgoing edges. One of them, p_3, has a single outgoing edge labeled *child* leading to a *title* vertex. Another instance, p_2, has two outgoing edges to *title* and *conference* vertices. Finally, p_1 has four outgoing edges. The lattice represents these three types of vertices with label *paper* separately, while a simple summary does not distinguish between them. Each node in lattice also stores the support of the *paper* vertex type it represents. We call the original summary a **level-0 summary** and the summary obtained by adding this lattice structure a **level-1 summary**. Using the level-1 summary, we can reason that there is no *paper* vertex in the database that connects to both *journal* and *conference* vertices, which is not possible using only level-0 summary. This process of enriching the summary by differentiating vertices based on the labels of their outgoing edges can be carried further by using the labels of vertices and edges that are reachable using paths of lengths two or more. We refer to such summaries as *level-k summaries*: A level-k summary differentiates vertices based on labels of edges and vertices on outgoing paths of length k. However, building level-k summaries for $k \geq 2$ is considerably more difficult than building level-0 and level-1 summaries. Level-0 summaries are essentially data guides, and level-1 summaries can be built with no additional cost if the file containing the graph edges is sorted by the identifiers of source vertices. For summaries of higher levels, additional passes of graph are required. Further, our experiments show that level-1 summaries are accurate enough for the datasets we study (Section 3), so the additional benefit of higher summary levels is unclear. In the rest of this paper, we focus on level-0 and level-1 summaries.

We assume that the graph database is stored on disk as a sequence of edges, sorted in lexicographic order of the source vertex. (Inputs in other formats, such as the ones used for the sample datasets in Section 3, are easily converted to this format.) Building level-0 and level-1 summaries requires only a single sequential scan of the edges file. We build the summary incrementally in memory as we scan the file. For an edge (v_1, v_2, l) we increment the counters associated with the summary nodes representing the labels l_1 and l_2 of v_1 and v_2, respectively. Similarly, the counter associated with the summary edge $(s(l_1), s(l_2), l)$ is incremented, where $s(l_i)$ denotes the summary node representing label l_i. (If the summary nodes or edges do not exist, they are created.) Since the edges file is sorted in lexicographic order of the source, we can be sure that we get all of the outgoing edges of a vertex before encountering another source vertex. Therefore, after processing all of the outgoing edges of a vertex during level-0 summary construction, we add an appropriate node to the corresponding lattice or increase the counter of an existing lattice node.

Algorithm *CandidateGeneration*($threshold$)
Input: The support threshold defining a frequent structure
Output: The set of all possible frequent structures

```
1.  candidate ←∅;
2.  open ←∅;
3.  for v ∈ summary and support(v) ≥ threshold
4.      do create a structure s consisting of a single vertex v;
5.         open ←open ∪s;
6.  while open ≠ ∅
7.      do S ←any structure in open;
8.         open ←open −S;
9.         candidate ←candidate ∪S;
10.        children ←expand(S);
11.        for c ∈ children
12.            do if support(c) ≥ threshold and c ∉ candidate
13.                then open ←open ∪c;
14. return candidate;
```

Figure 5: Simplified Candidate Generation Algorithm

We use a level-0 summary L_0 to estimate the support of a structure S as follows: By construction, there is at most one subgraph of L_0 (say, S') that is isomorphic to the summary of S. If no such subgraph exists, then the estimated (and actual) support of S is 0. Otherwise, let C be the set of counters on S' (i.e., C consists of counters on the nodes and edges of S'). The support of S is estimated by the minimum value in C. Given our construction of the summary, this estimate is an upper bound on the true support of S. With a level-1 summary L_1, we estimate the support of a structure S as follows: For each vertex v of S, let $L(v)$ be the set of lattice nodes in L_1 that represent a set of edges that is a superset of the set of out-edges of v. Let $c(v)$ denote the sum of the counters for nodes in $L(v)$. The support of S is estimated to be $\min_{v \in S} c(v)$. This estimate is also an upper bound on the true support of S. Further, it is a tighter bound than that given by the corresponding level-0 summary. For example, consider a structure consisting of a *paper* vertex with two out-edges, one to a *conference* vertex and the other to a *journal* vertex. Using the level-0 summary depicted in Figure 3, this structure's support is estimated as the minimum of the counters on the 3-node subgraph of the summary that is isomorphic to this structure's summary (lower left corner of the figure): $\min 1, 1, 3, 1, 1 = 1$. However, from the data graph in Figure 1, it is clear that the true support of this structure in 0. The level-1 summary estimates this support accurately at the expense of more book keeping. Section 3 presents an experimental evaluation of these estimates on real datasets.

2.2 Candidate Generation

A simplified version of our candidate generation algorithm is outlined in Figure 5: *CandidateGeneration(x)* returns a list of candidate structures whose estimated support is x or higher. It maintains two lists of structures: *open* and *candidate*. In the open list we store structures that have not been processed yet (and that will be checked later). The algorithm begins by adding all structures that consist of only one vertex and pass the support threshold test to the open list. The rest of the algorithm is a loop that repeats until there are no more structures to consider (i.e., the open list is empty.) In each iteration, we select a structure (S) from the open list and we use it to generate larger structures (called S's *children*) by calling the *expand* subroutine, described below (line 10). New child structures that have an estimated support of at least x are added to the open list. The qualifying structures are accumulated in the candidate result, which is returned as the output when the algorithm terminates.

Given a structure S, the *expand* subroutine produces the set of structures generated by adding a single edge to S (termed the children of S). In the following description of the *expand*(S) subroutine, we use $S(v)$ to denote the set of vertices in S that have the same label as vertex v in the data graph and $V(s)$ to denote the set of data vertices that have the same label as a vertex s in S. For each vertex s in S, we create the set *addable*(S, s) of edges leaving some vertex in $V(s)$. This set is easily determined from the data summary: It is the set of out-edges for the summary vertex representing s. (As we shall discuss in Section 3, this ability to generate structures using only the in-memory summary instead of the disk resident database results in large savings in running time.) Each edge $e = (s, v, l)$ in *addable*(S, s)

that is not already in S is a candidate for expanding S. If $S(v)$ (the set of vertices with the same label as e's destination vertex) is empty, we add a new vertex x with the same label as v and a new edge (s, x, l) to S. Otherwise, for each $x \in S(v)$ if (s, x, l) in not in S, a new structure is created from S and e by adding the edge (s, x, l) (an edge between vertices already in S). If s does not have an l-labeled edge to any of the vertices in $S(v)$, we also add a new structure which is obtained from S by adding a vertex x' with the same label as v and an edge (s, x', l).

For example consider the graph in Figure 1. Let us assume that we want to expand a structure S consisting of a single vertex s labeled *author*. The set *addable*(S, s) is $\{\mathrm{author} \stackrel{\mathrm{child}}{\longrightarrow} \mathrm{book}, \mathrm{author} \stackrel{\mathrm{idref}}{\longrightarrow} \mathrm{book}, \mathrm{author} \stackrel{\mathrm{child}}{\longrightarrow} \mathrm{name}, \mathrm{author} \stackrel{\mathrm{child}}{\longrightarrow} \mathrm{paper}\}$ (all the edges that leave an *author* labeled vertex in database). Since S has only one vertex, it can be expanded only by adding these four edges. Using the first edge in the addable set, a new structure is obtained from S by adding a new *book*-labeled vertex and connecting s to this new vertex by a *child* edge. The other edges in *addable*(S, s) give rise to three other structures in this manner.

2.3 Support Counting

Once the user is satisfied with the structures discovered in the candidate generation phase, she may be interested in finalizing the frequent structure list and getting the exact support of the structures. (Recall that the candidate generation phase provides only a quick, approximate support for each structure, based on the in-memory summary.) This task is performed in the support counting phase, which we describe here.

Let us define the size of a structure to be the number of nodes and edges it contains; we refer to a structure of size k as a *k-structure*. From the method used for generating candidates (Section 2.2), it follows that for every k-structure S in the candidate list there exists a structure S_p of size $k-1$ or $k-2$ in the candidate list such that S_p is a subgraph of S. We refer to S_p as the *parent* of S in this context. Clearly, every instance I of S has a subgraph I' that is an instance of S_p. Further, I' differs from I only in having one fewer edge and, optionally, one fewer vertex. We use these properties in the support counting process.

Determining the support of a 1-structure (single vertex) consists of simply counting the number of instances of a like-labeled vertex in the database. During the counting phase, we store not only the support of each structure (as it is determined), but also a set of pointers to that structure's instances on disk. To determine the support of a k-structure S for $k > 1$, we revisit the instances of its parent S_p using the saved pointers. For each such instance I, we check whether there is a neighboring edge and, optionally, a node that, when added to I generates an instance I' of S. If so, I' is recorded as an instance of S. This operation of growing an instance I of S_p to an instance I' of S is similar to the expand operation used in the candidate generation phase; however, there are two difference. First, in the counting phase we expand subgraphs of the database whereas in the candidate generation phase we expand abstract structures without referring to the disk-resident data (using only the summary). Second, in the counting phase we need to find an edge or vertex in the database to be added to the instance that satisfies the constraints imposed by the expansion which created the structure (e.g., the label of the edge). Whereas in the candidate generation phase, we add any possible edges and vertices to the structure.

A key operation in the above procedure is finding the edges and vertices in the database that potentially satisfy the expansion constraints for a given instance. Instead of scanning the database for this information (which would be very inefficient), for each vertex, we create an auxiliary file containing the out-edges of the vertex. These files are similar to path indexes and access support relations used in object and semistructured databases [KM90, M+98], and are created using a single pass through the database before the counting phase begins.

2.4 Quality-Speed Tradeoff

The SEuS method described above is significantly faster than other methods of which we are aware. (See Section 3.) Further, for applications that are not concerned with the exact supports of frequent structures (e.g., structured browsing, data exploration), the counting phase can often be skipped, resulting in interactive response times of a few seconds (between the input of parameters and the output of results). However, it presents only two options for the tradeoff between running time and the accuracy of supports: stopping at the search phase, in which case the estimate is an upper bound on the true support of a structure, and running the counting phase, in which case exact supports are produced. In this section, we present a method, called **L-SEuS**, that allows a user to tune the quality-speed tradeoff at a finer granularity.

Unlike SEuS, L-SEuS does not discover all frequent structures; instead, it returns only the *top-n* structures (n being an input parameter). By top-n, we mean the n structures that rank highest based on a scoring metric that we define below. We may be tempted to use support as the scoring metric, but doing so would result in structures consisting of a single node (1-structures) receiving the highest scores (since any k-structure with support t includes a number of 1-structures with supports no smaller than t). For example in the database of Figure 1, the most frequent structure would be the structure with a single vertex labeled *title*, because it has a support of 7. While technically correct by definition, it is very unlikely that such frequent structures are useful to an application because they convey very little information about the database. Therefore, we need a scoring metric that balances the support of a structure with its size. (An analogous problem does not occur in SEuS because it returns all frequent structures, not just the n highest-scoring ones.) For this purpose, we use the product of the support and size of a structure (where size refers to the sum of the number of vertices and number of edges in the structure); we refer to this metric as the structure's *score* below.

Another difference between SEuS and L-SEuS is the filtering scheme used in the candidate generation phase. In SEuS, we have a global threshold and every structure is compared with this threshold. This model suffers from the usual problem with absolute thresholds: If the user is not very familiar with the dataset (a likely situation in data exploration) then the system must be run with several guessed values of these thresholds before usable values are found. Since L-SEuS is an approximate method, in order to make it more suitable for preliminary explorations, we use a *local* comparison scheme in this method. In this model, a structure is frequent if its score is higher than the *adjusted score* of all of its children regardless of the score of other structures. The adjusted score $A(S)$ of a structure S is simply its score times a parameter called the *structure complexity measure (SCM)* ($0 \leq$ SCM ≤ 1): $A(S) = \text{support}(S) \times \text{size}(S) \times \text{SCM}$. A structure S is considered frequent if $\text{score}(S) > A(S')$ for all structures S' of which it is the parent structure (i.e., for all structures S' that can be generated from S by adding one edge and, optionally, one vertex).

In our work with the L-SEuS system and the datasets described in Section 3, we have found SCM to be a convenient tuning parameter. If we use large values for SCM, then we favor more complex structures and if we use SCM values near zero, the support becomes more important. An SCM value of one imposes no size constraints on the discovery process. It is implemented as a slider control in the L-SEuS system and allows us to quickly pick a value that returns structures that are both large and frequent enough. We are able to achieve interactive response times (essential for the slider control) because we return candidate structures generated by the search phase, which is very fast, and do not need to perform the counting phase.

These modifications only affect the candidate generation phase. (See Figure 5.) The termination condition on line 6 changes to incorporate early termination when we have found n structures. Also, the candidate set is updated (line 9) only if none of the children pass the *SCM test*: $\text{score}(c) \times \text{SCM} \geq \text{score}(s)$. On line 13, only the children passing this test should be added to the open list. The structure selection on line 7 has to be changed as well. Since this new method is greedy and will stop as soon as n structures are discovered, we should process the structures with higher scores first. Therefore, instead of choosing any arbitrary structure, we first process the structure with the highest score in the open list. In Section 3 we present experimental results studying the effects of these changes in semantics on the running time and solution quality.

L-SEuS is similar to the SUBDUE system in its functionality and solution strategy. Both methods return the top-n frequent structures based on some ranking metric. SUBDUE ranks structures using the compression ratio based on the Minimum Description Length (MDL) principle. (A higher scoring structure results in greater compression in a database encoding that replaces each instance of the structure with a node representing the structure, stored separately.) L-SEuS ranks structures using the score metric described above (size times support). L-SEuS's score metric is closely related to SUBDUE's compression ratio: It is simply the compression ratio when we ignore the space needed to store the mapping between the nodes that replace each instance of a structure and the structure itself. (L-SEuS can easily be modified to use the exact compression ratio as the ranking metric.) Finally, neither L-SEuS nor SUBDUE is complete; that is, they are greedy methods that are not guaranteed to return the most frequent structures. (The SEuS method is complete.)

2.5 Algorithm Analysis

The candidate generation process of SEuS uses an estimated support measurement. However, since this estimation is always higher than the actual value, SEuS will not miss any structure that might have a support higher than the threshold. This overestimation, on the other hand, might cause some structures with a support lower than the threshold to be added to the candidate list. Although these structures will be removed from the frequent structure list during the support count phase, the effort spent to count their support is wasted. We present an experimental study of this overestimation in Section 3.

On the other hand, our approximate method (L-SEuS) is not guaranteed to produce optimal solutions. (A solution is optimal if it consists of the n highest scoring structures.) We now discuss the three factors contributing to nonoptimal solutions in L-SEuS: the overestimation of a structure's support, the early termination condition, and the local comparison model.

Since L-SEuS is a greedy method and only returns the first n structures that it identifies as frequent, the order in which structures are considered for expansion is important. (Recall that the order of structure expansion is not important for correctness in SEuS because every structure will eventually be considered.) Structures are ordered based on their scores in the candidate generation phase of L-SEuS. A structure's score depends on its support which is estimated using the summary. Although estimates are guaranteed to be no lower than the actual supports, the amount of overestimation can vary across structures. For example, a structure whose estimate is a very large overestimate may cause a structure with higher score (but lower estimate) to be bumped off the candidate list.

As described in Section 2.4, L-SEuS uses a local model for comparing structures. Using this local model based on the SCM parameter has the benefits outline above; however it can also occasionally lead to some unintuitive results. A structure can be excluded from consideration solely based on a comparison with its parent. If a structure's parent has a very high score compared to all other structures, then excluding the child without comparing it with the rest of normal structure population might not be fair. As an example, suppose s_1 and s_2 are two structures being considered for expansion in different iterations of the candidate generation phase with SCM = 1. Further, suppose $\text{support}(s_1) = 1000, \text{support}(s_2) = 50$, and $\text{size}(s_1) = \text{size}(s_2) = 3$. If structures s_{11} and s_{21} resulting from expanding s_1 and s_2 both have a support equal to 40 and $\text{size}(s_{11}) = \text{size}(s_{21}) = 5$ (one additional edge and vertex) then s_{21} will be added to the open list while s_{11} will not even be considered. This decision is made despite the fact that $\text{score}(s_{11}) = \text{score}(s_{21})$. This inconsistency occurs because we use a local comparison model rather than a global one. This problem affects only the L-SEuS (and not SEuS) method.

3 Experimental Evaluation

In order to evaluate the running time of our method and the quality of the solutions it produces, we have performed a number of experiments. We have implemented SEuS and L-SEuS using the Java 2 (J2SE) programming environment. For graph isomorphism tests, we have used the *nauty* package[McK02] to derive canonically labeled isomorphic graphs. Since we have two levels of summaries, we append a "-Sd" to a system's name to show which level of summary has been used with the method in a particular experiment (e.g., SEuS-S0 is the SEuS method using summary level-0). In the experiments described below, we have used a PC-class machine with a 900 MHz Intel Pentium III processor and one gigabyte of RAM, running the RedHat 7.1 distribution of GNU/Linux. Where possible, we have compared our results with those for SUBDUE version 4.3 (serial version), which is implemented in the C programming language. Table 2 presents some characteristics of the 13 datasets we have used for our experiments, with references to their sources.

Figure 6 compares the running time of SEuS, L-SEuS, and SUBDUE on the 13 datasets of Table 2. Running times of SEuS and L-SEuS using both levels of summaries are depicted here. It is important to notice that SEuS versions run for a longer time because they are looking for all frequent structures, whereas L-SEuS and SUBDUE only return the n most frequent structures ($n = 5$ in these experiments.). The running times of SEuS and L-SEuS increase monotonically as the size of datasets increases. The irregularities in the running time of SUBDUE are due to the fact that, besides the size of a dataset, factors such as the number of vertex and edge labels have a significant effect on the performance of SUBDUE. Referring to Table 2, it is clear that *Credit* datasets have many more labels than the *Diabetes* datasets. Although *Credit-1* and *Credit-2* datasets are smaller than the *Diabetes* datasets, it takes SUBDUE longer to mine them because it tries to expand the subgraphs by all possible edges at each iteration. Then SUBDUE decides which isomorphism class is better by considering the number of subgraphs in them and the size of the subgraphs. (In SUBDUE the sets of isomorphic subgraphs are manipulated as bags of subgraphs.) When there is a large number of different vertex or edge labels, there will be a larger number of subgraphs to choose between and since SUBDUE accesses the database for each subgraph, the running time increases considerably. The number of edge or vertex labels affects SEuS and L-SEuS in a similar way, but since we do not access the main database to find the support of a structure (we use the summary instead) this number does not significantly affect our running time.

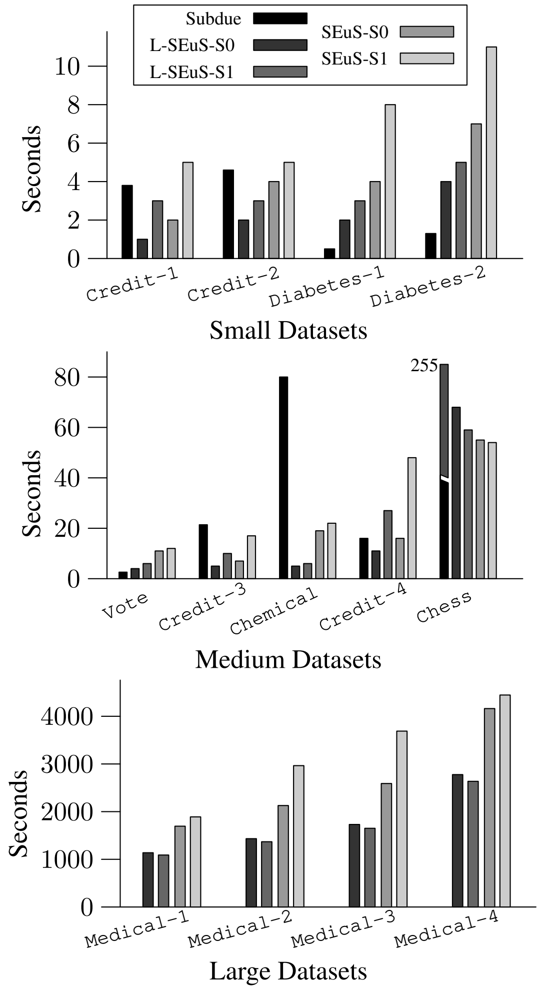

Figure 6: Running time

SEuS and L-SEuS have a phase of data summary generation which SUBDUE does not perform. In small datasets this additional effort is comparable to the overall running time. For example, while running on *Diabetes* datasets, L-SEuS-S0 takes longer than SUBDUE mainly because of the summary generation overhead. Later in this section we show that as the dataset size grows this overhead becomes negligible. Also note that the running time of SEuS and L-SEuS increases if we use level-1 summary instead of level-0 summary. This increase in running time is mainly due to the overhead of creating a richer summary. Later, we will see that this additional effort will result in more accurate results. We are comparing a Java implementation of (L-)SEuS with the C implementation of SUBDUE. While the difference in efficiency of these programming environments is not significant for large datasets, it is a factor for the smaller ones.

As the datasets grow, the running time of SUBDUE grows very quickly, while (L-)SEuS does not show such a sharp increase. With our experimental setup, we were unable to obtain any results from SUBDUE for datasets larger than 3 MB (after running for 24 hours). For this reason, Figure 6 presents the running time of only SEuS and L-SEuS methods for the large datasets. To best of our knowledge, other complete structure discovery methods cannot handle datasets with sizes comparable to those we have used here. As mentioned earlier, the AGM and FSG methods take respectively eight days and 600 seconds to process the *Chemical* dataset, for which SEuS only needs 20 seconds[KK01]. (Unfortunately, we were unable to obtain the FSG system to perform a more detailed comparison.) One should note that for very small thresholds, these methods will have a better performance because with those thresholds a large number of structures will be frequent and our summary does not provide a significant pruning while introducing the overhead of creating a summary.

Dataset	Size	Average percentage of overestimation		Total number of cases
		Level-0	Level-1	
Credit-1	3899	1.38	0.00	55
Credit-2	3899	0.50	0.00	48
Diabetes-1	4556	5.78	0.00	58
Diabetes-2	8500	3.39	0.03	47
Vote	8811	1.62	0.00	50
Credit-3	12300	1.60	0.00	49
Chemical	18506	0.00	0.00	34
Credit-4	27800	0.30	0.00	48
Chess	189311	0.00	0.00	14
Medical-1	3999997	0.17	0.00	43
Medical-2	4999997	0.15	0.00	40
Medical-3	5999997	0.16	(0.00)	42
Medical-4	9529355	0.18	0.00	33

Table 1: Average overestimate of support of the structures

In the experiments studying running time, we have used a fixed SCM value of 0.9 for L-SEuS methods (which, recall, return only the top-n structures). However, a similar strategy of using a fixed threshold (absolute or relative to database size) is impractical for the SEuS methods, which return all frequent structures. We found that a support threshold that returns a reasonable number of structures for one dataset results in far too many for another. Raising the threshold to fix the problem with the second dataset results in no frequent structures for the first. (This experience exemplifies the need for an interactive system which gives a user rapid feedback to enable selection of parameters based on the characteristics of a dataset.) For the SEuS methods, we used this interactive procedure to select threshold values that result in roughly 50 frequent structures for each dataset. (These thresholds are mentioned in Table 3.) As Figure 6 shows, for large datasets the L-SEuS methods are faster than the SEuS methods. However, SEuS methods take at most twice as long as the approximate L-SEuS methods. This result supports our suggested strategy for exploring datasets of this magnitude: Use L-SEuS method (with or without the counting phase) for initial interactive explorations in order to select proper thresholds that can then be used with SEuS to get accurate results.

Recall from Section 2.1 that we use an estimated support for structures in the course of discovery and that this estimation never underestimates the actual support. In Table 1, we summarize the average overestimation on the test datasets using level-0 and level-1 summaries. The zero entries for level-1 summary have an absolute zero average overestimation percentage (except *Medical-3* which is rounded to 0.0). As the number of structures for each dataset indicates, in these experiments our threshold values were relatively high (resulting in roughly 50 structures). Therefore most of the structures are small. (e.g., for *Credit-4* dataset, the maximum size of the discovered structures is 7.) In these small structures, an overestimation of absolute zero is reasonable. However, one should note that the overestimation will increase as the structures grow larger. This table indicates that using a level-1 summary gives us sufficient accuracy while saving a lot of effort (compared with using level-k summaries for $k > 1$ or to not using summaries).

Recall that L-SEuS and SUBDUE are not complete methods. They return only n frequent structures, ranked by the metrics described in Section 2.4. We performed a series of experiments to evaluate the quality of structures returned by L-SEuS and SUBDUE. The metric for structure quality is, in general, domain dependent. However, for the purposes of an objective evaluation in this paper, we use the *compression gain* metric. The score metric described in Section 2.4 is an efficient approximation of the compression gain and we use it for the quality metric here. More precisely, let $c(S)$ and $o(S)$ denote, respectively, the size and support of a structure S. Let $(B_i)_{i=1}^{n}$ denote the top-n structures based on the scoring function $f(S) = c(S)o(S)$. Let $(G_i)_{i=1}^{n}$ denote the top-n structures as produced by a method (L-SEuS or SUBDUE). We use the *compression loss* of a method, defined as $(\sum_{i=1}^{n} f(B_i) - \sum_{i=1}^{n} f(G_i))/\sum_{i=1}^{n} f(B_i)$, to measure the loss in quality of the method (compared with the optimal solution). In order to get the optimal solution, we run the complete SEuS method (with parameters set to return *all* structures) on these datasets and then rank the structures based on their score. In Figure 7, we compare the percentage compression loss for L-SEuS and SUBDUE. We have plotted the compression loss for five most frequent structures ($n = 5$). Combinations of methods and datasets with no bar signifies zero compression loss. As this figure shows, for most of the datasets SUBDUE performs worse

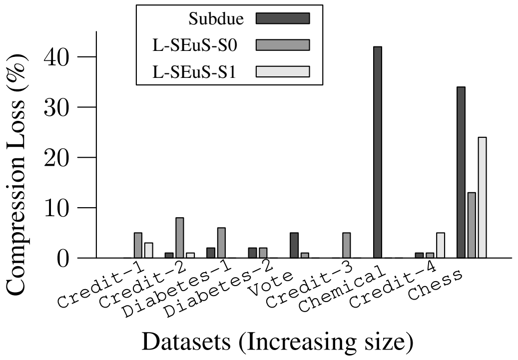

Figure 7: Compression Loss Comparison

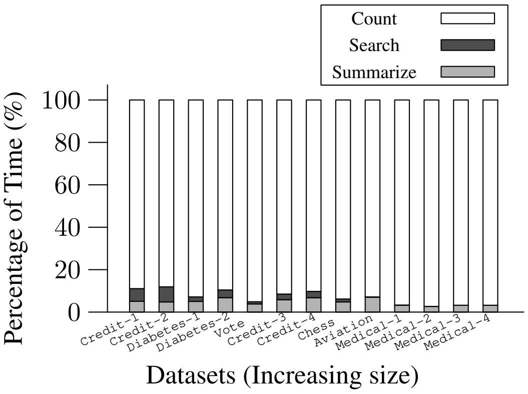

Figure 8: Time spent in algorithm phases

than L-SEuS and for the *Chemical* dataset, SUBDUE's error percentage is very high. One might expect that L-SEuS-S1 should always have a better quality. If L-SEuS were not a greedy algorithm this assumption would be a correct. However, since the method stops as soon as n structures are found, there is no guarantee that a better estimate will produce results with higher scores.

As discussed in Section 1, the SEuS system provides real-time feedback to the user by quickly displaying the frequent structures resulting from different choices of the threshold (or SCM) parameter. This interactive feedback is possible because the time spent in the candidate generation (search) phase is very small. Figure 8 justifies this claim. It depicts the percentage of time used by each of the three phases in processing different datasets. As datasets get larger, the fraction of running time spent on summarizing the graph falls rapidly. Also the time spent in the candidate generation phase is relatively small. Therefore, our strategy of creating the summary once and running the candidate generation phase multiple times with different input parameters (in order to determine suitable values before proceeding to the expensive counting phase) is very effective.

It is important to note that the counting phase is performed only to find the exact support of the structures. This step is necessary if the output of this system is to be used as input for a more complex mining method or if the user wishes to know the exact supports. On the other hand, if user is interested primarily in the structures themselves, and not the exact supports, the counting phase can be completely skipped. Skipping this phase does not affect the actual structures produced as output; therefore the quality of the structures remains the same while the running time decreases substantially.

In the appendix, we present some additional experimental results. Table 4 summarizes the sensitivity of the running time of SEuS to the support threshold parameter. (Note that in worst case, the size of the output grows to a size exponential in the size of the input database as threshold is lowered; thus it is unavoidable that all methods that produce a complete output, such as SEuS, will experience a rapid rise in running time with falling thresholds.) In Figures 9–14, we summarize the effect of the SCM parameter on the running time of L-SEuS. As expected, running time tends to rise as the threshold value increases; however, the increase is not very dramatic, supporting the suitability of L-SEuS for preliminary exploratory data analysis.

4 Related Work

Much of the prior work on structure discovery is domain dependent (e.g., [Win75, Lev84, Fis87, Leb87, GLF89, CG92]) and a detailed comparison of these methods appears in [Con94]. We consider only domain independent methods in this paper. The first such system, CLIP, discovers patterns in graphs by expanding and combining patterns discovered in previous iterations [YMI93]. To guide the search, CLIP uses an estimate of the compression resulting from an efficient representation of repetitions of a candidate structure. The estimate is based on a linear-time approximation for graph isomorphism. SUBDUE [CH00] also performs structure discovery on graphs. It uses the minimum description length principle to guide its beam search. SUBDUE uses an inexact graph matching algorithm during the process to find similar structures.

SUBDUE discovers structures differently from CLIP. First, SUBDUE produces only single structures evaluated using minimum description length, whereas CLIP produces a set of structures that collectively compress the input graph. CLIP has the ability to grow structures using the merge operator between two previously found structures, while SUBDUE only expands structures one edge at a time. Our system is similar to SUBDUE with respect to structure expansion. Second, CLIP estimates the compression resulting from using a structure, but SUBDUE performs an expensive exact measurement of compression for each new structure. This expensive task causes the SUBDUE system to be very slow when operating on large databases, because for each new concept discovered, the system goes through the input graph and calculates the gain in compression using this new structure. The issues of scaling the SUBDUE system and implementing the method in a parallel environment have been addressed in [CH$^+$01], which presents three approaches to distribute the original algorithm in a parallel environment.

AGM [IWM00] is an Apriori-based algorithm for mining frequent structures. The main idea is similar to that used by the market basket analysis algorithm in [AS94]: a $(k+1)$-itemset is a candidate frequent itemset only if all of its k-item subsets are frequent. In AGM, a graph of size $k+1$ is considered to be a candidate frequent structure only if all its subgraphs of size k are frequent. In AGM, only the *induced* subgraphs are considered to be candidate frequent structures. (Given a graph G, subgraph G_s is called an induced subgraph if $V(G_s) \subset V(G), E(G_s) \subset E(G)$ and $\forall u, v \in V(G_s), (u,v) \in E(G_s) \Leftrightarrow (u,v) \in E(G)$.) This restriction reduces the size of the search space, but also means that interesting structures that are not induced subgraphs cannot be detected by AGM. After producing the next generation of candidate frequent structures, AGM counts the frequency of each candidate by scanning the database. As in SUBDUE, this need for a database scan at each generation limits the scalablity of this method. In contrast, our method is not limited to induced subgraphs and does not scan the database at each generation.

FSG [KK01] is another system that finds all connected subgraphs that appear frequently in a large graph database. Similar to AGM, this system uses the level-by-level expansion adopted in Apriori. The key features of FSG compared to AGM are the following: (1) it uses a sparse graph representation which minimizes storage and computation, (2) there is no restriction on the structure's topology (e.g., induce subgraph restriction) other than their connectivity, and (3) it incorporates a number of optimizations for candidate generation and counting which makes it more scalable (e.g., transaction ID lists for counting). However, this system still scans the database in order to find the support of next generation structures. The experimental results in [KK01] show that FSG is considerably faster than AGM. One should note that AGM and FSG both operate on a transaction database where each transaction is a graph, so that their definition of a frequent structure's support can be applicable. In SEuS we do not have this restriction, and SEuS can be applied to both a transaction database and a large connected graph database. As mentioned in Section 3, for a common *Chemical* dataset, FSG needs 600 seconds, where SEuS returned the frequent structures in less than 20 seconds.

The problem of finding frequent structures is related to the problem of finding implicit structure (or approximate typing) in semistructured databases. In [NAM97], the authors propose a method to infer an approximate classification of objects into a hierarchical collection of types. This method uses a counting lattice (similar to our level-1 summary) to summarize the graph database and then use a heuristic function called *jump* to identify the candidate types. It then builds a type hierarchy based on these candidates and infers the typing rules. In [NAM98], the authors address the same problem using the greatest fixpoint semantics of monadic datalog programs. First, they define a type for each object in the database using a datalog program. Then, they use a technique similar to k-clustering to merge similar types, until there are n types left. The merge is done based on a distance function defined between the datalog programs representing two types. These papers do not present a detailed performance analysis and it is not clear how these methods would scale to large datasets such as those on which we focus in this paper.

Furthermore, there are important differences between the problems of type inference and frequent structure discovery. In type inference, the structures are typically limited to rooted trees and each structure must have a depth of one. Further, the frequency of a structure is not the only metric used in type inference. For instance, a type that occurs infrequently may be important if its occurrences have a very regular structure. Despite these differences, it may be interesting to investigate the possibility of adapting methods from one problem for the other.

5 Conclusion

In this paper, we motivated the need for data mining methods for large semistructured datasets (modeled as labeled graphs with several million nodes and edges). We focused on an important building block for such data mining methods: the task of finding frequent structures, i.e., structures that are isomorphic to a large number of subgraphs of the input graph. We presented the SEuS method, which finds frequent structures efficiently by using a three-phase approach: The first phase builds a structural summary, the second uses this summary to generate candidate frequent structures, and the third generates the frequency of each structure. We also presented a faster, approximate method, L-SEuS, which returns approximate results suitable for rapid exploratory analysis. We have implemented these methods in the SEuS system for exploring semistructured data; our implementation is freely available under GNU GPL terms. We presented the results of a detailed experimental study of the running time of SEuS and L-SEuS and the quality of the approximate solutions produced by L-SEuS.

Our methods have two main distinguishing features: First, due to their use of a summary data structure, they can operate on datasets that are two to three orders of magnitude larger than those used by prior work. Second, our methods provide rapid early feedback (delay of a a few seconds) in the form of candidate structures, thus permitting their use in an interactive data exploration system. We have found this rapid feedback from the first two stages of our methods to be invaluable in selecting a suitable value for support threshold parameter. Further, in some applications (such as our system for data exploration), the third (and most time-consuming) phase of our methods can be skipped since we are interested in only the qualitative characteristics of frequent structures, not their exact frequencies.

As ongoing work, we are exploring the application of our methods to finding association rules and other correlations in semistructured data. We are also applying our methods to the problems of classification and clustering by using frequent structures to build a predictive model.

Acknowledgments

We would like to thank the SUBDUE team for providing us with the SUBDUE code and some of the datasets used in the experiments. We would also like to thank the UCI repository of machine learning, the University of York machine learning group, National Library of Medicine, and Oxford University Computing Laboratory for the datasets. The complexity analysis of finding the support of the most frequent k-structure is due to Aravind Srinivasan.

References

[AIS93] R. Agrawal, T. Imielinski, and A. Swami. Mining associations between sets of items in massive databases. *SIGMOD Record*, 22(2):207–216, June 1993.

[AS94] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. of the 20th International Conference Very Large Data Bases*, pages 487–499. Morgan Kaufmann, 1994.

[BDFS97] P. Buneman, S. B. Davidson, M. F. Fernandez, and D. Suciu. Adding structure to unstructured data. In *Proc. of the 6th International Conference on Database Theory*, 1997.

[BM] C.L. Blake and C.J. Merz. UCI repository of machine learning databases.

[CG92] D. Conklin and J. Glasgow. Spatial analogy and subsumption. In *Proc. of the Ninth International Conference on Machine Learning*, pages 111–116, 1992.

[CH00] D. J. Cook and L. B. Holder. Graph-based data mining. *ISTA: Intelligent Systems & their applications*, 15, 2000.

[CH$^+$01] D. J. Cook, L. B. Holder, et al. Approaches to parallel graph-based knowledge discovery. *Journal of Parallel and Distributed Computing*, 2001.

[Con94] D. Conklin. Structured concept discovery: Theory and methods. Technical Report 94-366, Queen's University, 1994.

[Fis87] D. H. Fisher, Jr. Knowledge acquisition via incremental conceptual clustering. *Machine Learning*, (2):139–172, 1987.

[For96] S. Fortin. The graph isomorphism problem. Technical Report 96-20, University of Alberta, 1996.

[GLF89] J. H. Gennari, P. Langley, and D. Fisher. Models of incremental concept formation. *Artificial Intelligence*, (40):11–61, 1989.

[GW97] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *Proc. of the Twenty-Third International Conference on Very Large Data Bases*, pages 436–445, 1997.

[IWM00] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *Proc. of the 4th European Conference on Principles and Practice of Knowledge Discovery in Databases*, pages 13–23, 2000.

[KK01] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *Proc. of the 1st IEEE Conference on Data Mining*, 2001.

[KM90] A. Kemper and G. Moerkotte. Access support in object bases. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 364–374, 1990.

[Leb87] M. Lebowitz. Experiments with incremental concept formation: Unimem. *Machine Learning*, (2):103–138, 1987.

[Lev84] R. Levinson. A self-organizing retrieval system for graphs. In *Proc. of the National Conference on Artificial Intelligence*, pages 203–206, 1984.

[M$^+$98] J. McHugh et al. Indexing semistructured data. Technical report, Stanford University, Computer Science Department, 1998.

[McK02] B. D. McKay. nauty user's guide (version 1.5), 2002.

[Med01] Medical citation databases, 2001.

[NAM97] S. Nestorov, S. Abiteboul, and R. Motwani. Inferring structure in semistructured data. In *Proc. of the Workshop on Management of Semistructured Data*, 1997.

[NAM98] S. Nestorov, S. Abiteboul, and R. Motwani. Extracting schema from semistructured data. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 295–306, 1998.

[NUWC97] S. Nestorov, J. Ullman, J. Wiener, and S. Chawathe. Representative objects: Concise representations of semistructured, hierarchial data. In *Proc. of the International Conference on Data Engineering*, pages 79–90, 1997.

[Oxf97] The predictive toxicology evaluation challenge datasets, 1997.

[Win75] P. H. Winston. Learning structural descriptions from examples. In *The Psychology of Computer Vision*, pages 157–209. 1975.

[YMI93] K. Yoshida, H. Motoda, and N. Indurkhya. Unifying learning methods by colored digraphs. In *Proc. of the International Workshop on Algorithmic Learning Theory*, volume 744, pages 342–355, 1993.

[Yor] University of York machine learning group.

Name	Description	Vertices	Edges	Vertex labels	Edge labels	Graph Size	Summary Size
Credit-1	Credit card application db [BM]	1999	1900	59	20	3899	136
Credit-2	Credit card application db [BM]	1999	1900	58	20	3899	134
Diabetes-1	Diabetes patient records [BM]	2412	2144	7	8	4556	39
Diabetes-2	Diabetes patient records [BM]	4500	4000	7	8	8500	38
Vote	Congressional voting records [BM]	4539	4272	4	16	8811	52
Credit-3	Credit card application db [BM]	6300	6000	59	20	12300	136
Chemical	Chemical compounds [Oxf97]	9189	9317	66	4	18506	338
Credit-4	Credit card application db [BM]	14700	14000	59	20	27800	137
Chess	Chess relational domain [Yor]	76272	113039	7	12	189311	88
Medical-1	Medical publication citations [Med01]	1999999	1999998	75	4	3999997	175
Medical-2	Medical publication citations [Med01]	2499999	2499998	75	4	4999997	174
Medical-3	Medical publication citations [Med01]	2999999	2999998	75	4	5999997	177
Medical-4	Medical publication citations [Med01]	4764678	4764677	75	4	9529355	190

Table 2: Datasets used in experiments

	Threshold (Percentage of graph size)						
Dataset	Credit-1	Credit-2	Diabetes-1	Diabetes-2	Vote	Chemical	Chess
SEuS-S0	3.4	3.9	5.8	6.1	4.4	1.5	3.7
SEuS-S1	2.8	3.1	3.0	3.7	3.8	1.2	3.6
Dataset	Credit-3	Credit-4	Medical-1	Medical-2	Medical-3	Medical-4	
SEuS-S0	3.5	3.9	2.5	2.5	2.5	2.7	
SEuS-S1	2.9	3.1	1.0	1.0	1.0	1.5	

Table 3: Thresholds used to generate roughly 50 frequent structures

Appendix

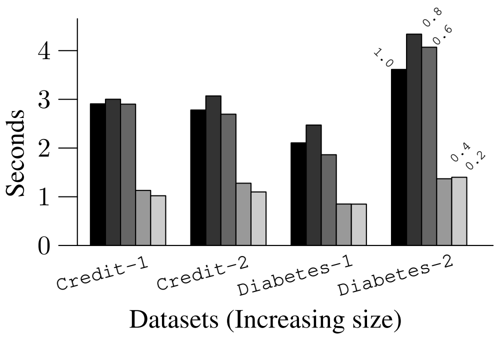

Figure 9: L-SEuS-S0 running time sensitivity to SCM - small datasets

Dataset	Thresholds (Percentage of graph size)						
	20	10	5	2	1	0.7	0.5
Credit-1	1029	1096	1104	26779	53016		
Credit-2	1063	1079	1084	45520	54109		
Diabetes-1	1365	1405	2745	18517	57300		
Diabetes-2	2309	2330	5905	42222	100152		
Vote	2234	2333	2700	111512			
Credit-3	3389	3405	3287	101768	186106		
Chemical	3460	7416	9122	9221	31801	42099	94050
Credit-4	10410	10605	10914	291315	431767		
Chess	40274	40951	44735	682426			
Medical-1	613151	858222	1519327	1973250	19463120		
Medical-2		1044442	1849024	2462402	23672410		
Medical-3		1332238	2405657	2954496	29269385		
Medical-4		1975776	3553597	4735045	47266950		

Table 4: SEuS running time sensitivity to threshold (in milliseconds)

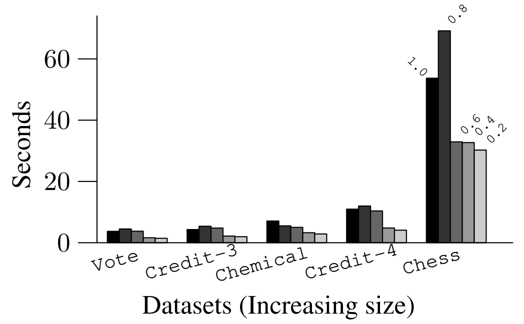

Figure 10: L-SEuS-S0 running time sensitivity to SCM - medium datasets

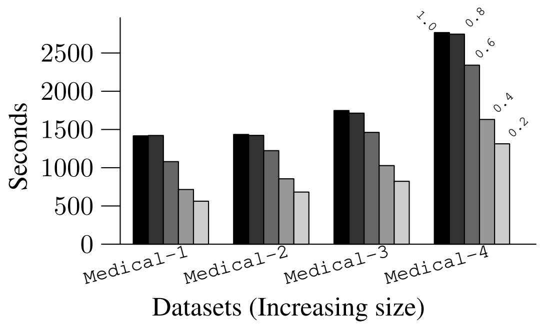

Figure 11: L-SEuS-S0 running time sensitivity to SCM - large datasets

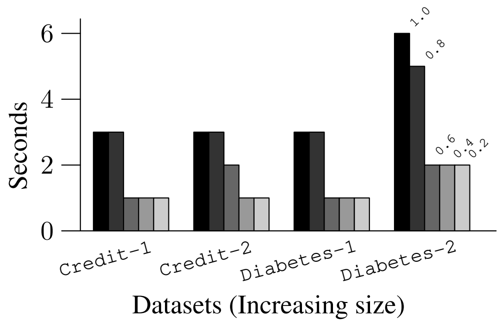

Figure 12: L-SEuS-S1 running time sensitivity to SCM - small datasets

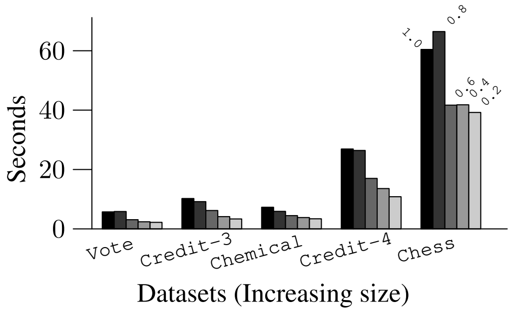

Figure 13: L-SEuS-S1 running time sensitivity to SCM - medium datasets

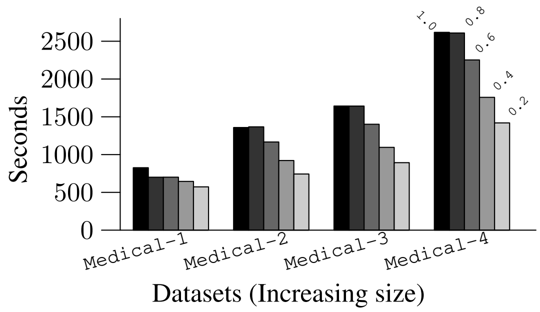

Figure 14: L-SEuS-S1 running time sensitivity to SCM - large datasets

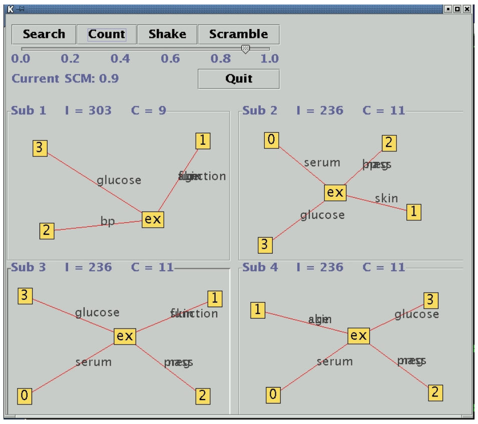

Figure 15: A snapshot of the L-SEuS after candidate generation phase