SEuS: Structure Extraction using Summaries

Shayan Ghazizadeh and Sudarshan S. Chawathe

University of Maryland
Department of Computer Science
College Park, MD
{shayan,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 is capable of operating on datasets that are two to three orders of magnitude larger than those described in prior work.

1 Introduction

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. Semistructured data is referred to data who has some structure, but is difficult to describe with a predefined, rigid schema. The structure of semistructured data is irregular, incomplete, frequently changing, and usually implicit or unknown to user. Common examples of this type of data include memos, Web pages, documentation, and bibliographies.

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* 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. Our summary is similar to data guides and other (approximate) typing mechanisms for semistructured data [GW97, BDFS97, NUWC97, NAM97]. 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. 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*).

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. Due to space limitations we are not able to present the details of this approximate method here. Interested readers can refer to [GC02].

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. Figure 1 is a screenshot of our system in action. The current set of frequent structures is displayed together with a slider that allows the threshold to be modified. Given a

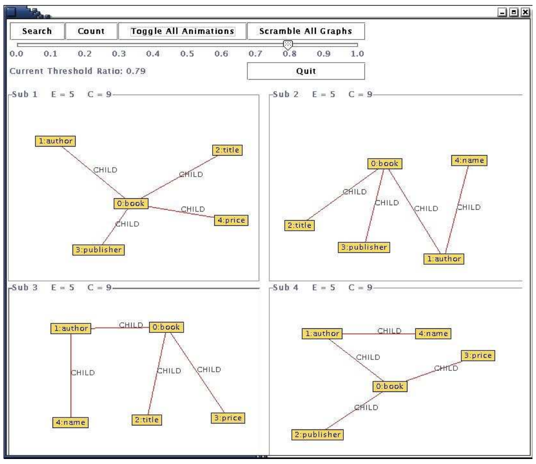

Fig. 1. A snapshot of SEuS system

new value for the threshold, the system computes (in interactive times) the new set of frequent structures and presents them as depicted. We have found this iterative process to be very effective in arriving at interesting values of the parameter.

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 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 2 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 2, a structure and its three instances are shown in Figure 3. 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

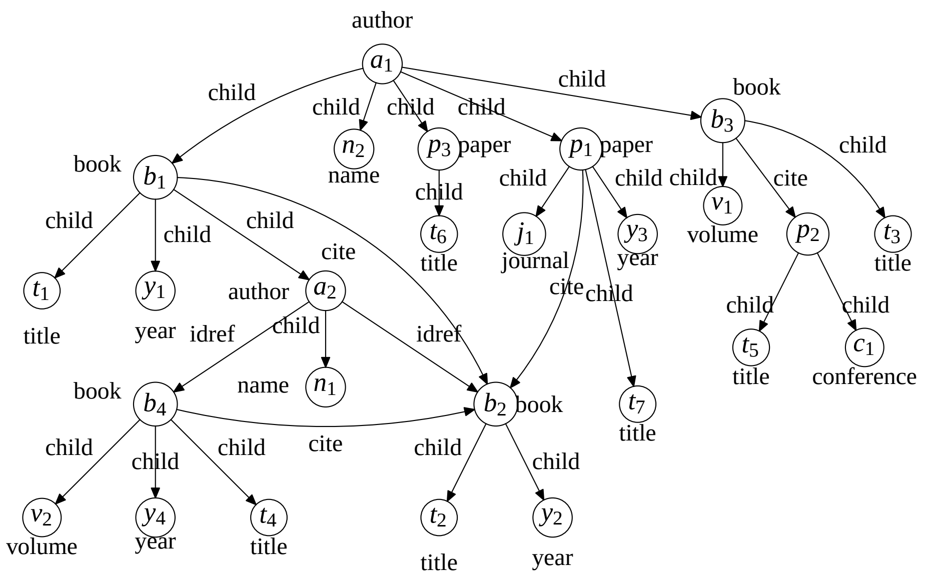

Fig. 2. Example input graph

isomorphism in worst case. 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.

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.

A number of systems have used such a strategy for discovering frequent structures [IWM00, KK01, CH00] along other heuristics to speed up the process. (See Section 4 for details.) However, The results reported in these papers, as well as our experiments, suggest that these methods do not scale to very large databases.

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 and other heuristics), 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

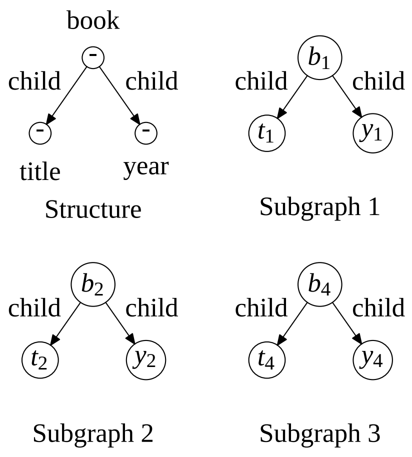

Fig. 3. A structure and its three instances

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 4 depicts the summary generated for the input graph of Figure 2.

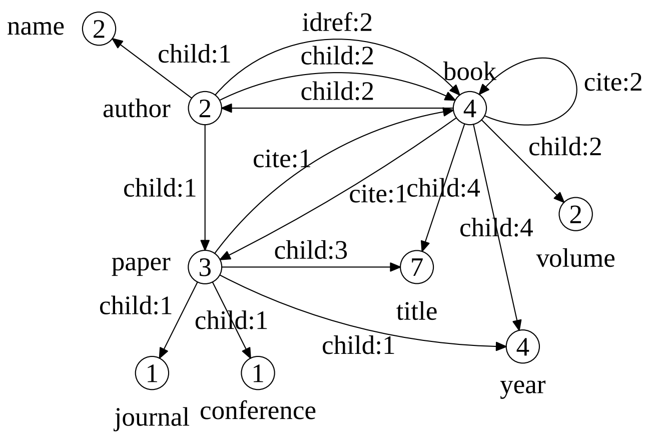

Fig. 4. Summary graph

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 2 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 1 in Section 3.) 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.)

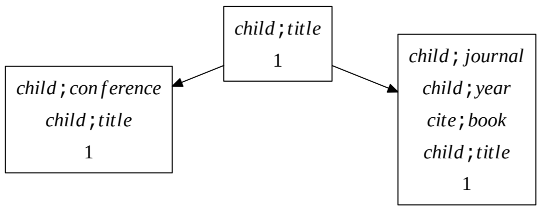

Fig. 5. Counting Lattice for *paper* vertex

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 of outgoing edges. For example, consider the vertex label *paper* in Figure 2. The counting lattice for this vertex is depicted in Figure 5. 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 (See [GC02] for details), 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. Clearly, building level-0 summary 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, we can create the level-1 summary in the same pass as we build the level-0 summary.

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 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.

2.2 Candidate Generation

A simplified version of our candidate generation algorithm is outlined in Figure 6: *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. 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.

```
Algorithm CandidateGeneration($threshold$)
1.  candidate $\leftarrow\emptyset$; open $\leftarrow\emptyset$;
2.  for $v \in$ summary and support($v$) $\geq$ threshold
3.      do create a structure $s$ consisting of a single vertex $v$;
4.         open $\leftarrow$open $\cup s$;
5.  while open $\neq \emptyset$
6.     do $S$ $\leftarrow$any structure in open;
7.        open $\leftarrow$open $-S$; candidate $\leftarrow$candidate $\cup S$;
8.        children $\leftarrow$expand($S$);
9.        for $c \in$ children
10.           do if support($c$) $\geq$ threshold and $c \notin$ candidate
11.                then open $\leftarrow$open $\cup c$;
```

Fig. 6. Simplified Candidate Generation Algorithm

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 2. 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 $\{\text{author} \overset{\text{child}}{\longrightarrow} \text{book}, \text{author} \overset{\text{idref}}{\longrightarrow} \text{book}, \text{author} \overset{\text{child}}{\longrightarrow} \text{name}, \text{author} \overset{\text{child}}{\longrightarrow} \text{paper}\}$ (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.

3 Experimental Evaluation

In order to evaluate the performance of our method we have performed a number of experiments. We have implemented 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. Due to space restictions we are not able to present detailed experimental results here. Extensive results have been presented in [GC02]. Table 1 presents some characteristics of the 13 datasets we have used for our experiments.

Name	Description	Vertex labels	Edge labels	Summary Size	Graph Size
Credit-*	Credit card applications	59	20	136	3899-27800
Diabetes-*	Diabetes patient records	7	8	39	4556-8500
Vote	Congressional voting records	4	16	52	8811
Chemical	Chemical compounds	66	4	338	18506
Chess	Chess relational domain	7	12	88	189311
Medical-*	Medical publication citations	75	4	175	4M-10M

Table 1. Datasets used in experiments

Figure 7 compares the running time of SEuS and SUBDUE on the 13 datasets of Table 1. Running times of SEuS using both levels of summaries are depicted here. It is important to notice that SEuS versions run for a longer time compared to SUBDUE because it is looking for all frequent structures, whereas SUBDUE only returns the n most frequent structures ($n = 5$ in these experiments.). The running times of SEuS increases 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 1, 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 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.

SEuS has a phase of data summary generation which SUBDUE does not perform. In small datasets this additional effort is comparable to the overall running time. Also note that the running time of 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. This additional effort will result in more accurate results (lower over-estimation which yeilds less waste of time in the counting phase). We are comparing a Java implementation of 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.

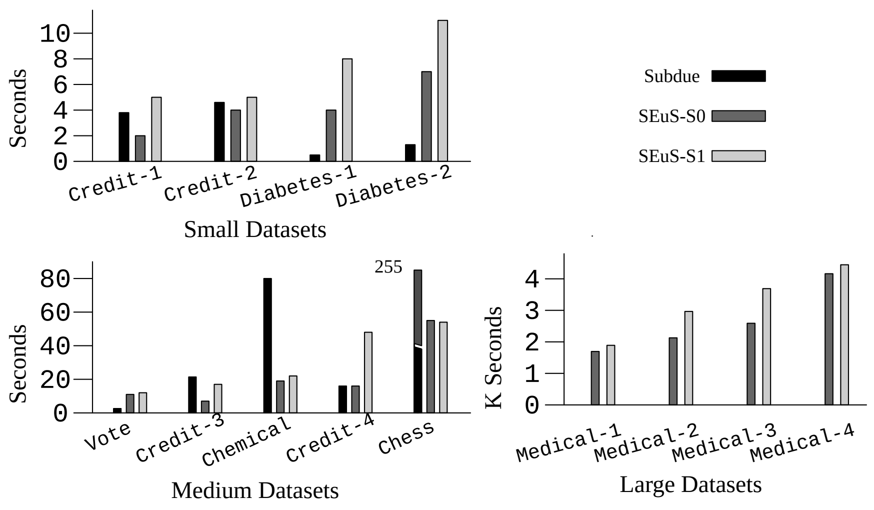

Fig. 7. Running time

As the datasets grow, the running time of SUBDUE grows very quickly, while 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 7 presents the running time of only SEuS method 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. AGM and FSG methods, presented in [IWM00, KK01], 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.

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 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.

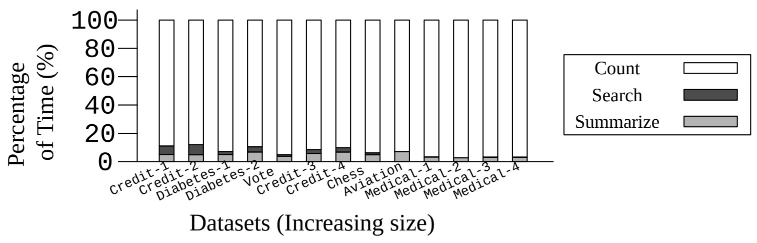

Fig. 8. Time spent in algorithm phases

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.

AGM [IWM00] is an Apriori-based algorithm for mining frequent structures which are induced subgraphs of the input graph. 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. AGM only considers the *induced* subgraphs 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.

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.

Asai [AAK$^+$02] proposes FREQT algorithm for discovering frequent structures in semistructured data. FREQT models semistructured data and the frequent structures using labeled ordered trees. The key contribution of this work is the notion of the rightmost expansion, a technique to grow a tree by attaching new nodes only to the rightmost branch of the tree. The authors show that it is sufficient to maintain only the instances of the rightmost leaf to efficiently implement incremental computation of structure frequency. Limiting the search space to ordered trees allows the method to scale almost linearly in the total size of maximal tree contained in the collection.

In [CYLW02], authors propose another method for frequent structure discovery in semistructured collections. In this work, the dataset is a collection of semistructured objects treated as transactions similar to FSG method. Motivated by the semistructured data path expressions, the authors try to represent the objects and patterns as a set of labeled paths which can include wildcards. After introducing the notion of *weaker than* for comparing a structure path set with a transaction object, the algorithm tries to discover the set of all patterns that have a frequency higher than a given threshold. The authors discuss that the methods is motivated and well-suited for collections consisting of similarly structured objects with minor differences.

The problem of finding frequent structures is related to the problem of finding implicit structure (or approximate typing) in semistructured databases [NAM97, NAM98]. 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 structural summary to estimate structure support.

Our method has 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 feedback (delay of a few seconds) in the form of candidate structures, thus permitting their use in an interactive data exploration system.

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.

References

[AAK$^+$02] Tatsuya Asai, Kenji Abe, Shinji Kawasoe, et al. Efficient substructure discovery from large semi-structured data. In *Proc. of the Second SIAM International Conference on Data Mining*, 2002.

[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.

[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.

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

[CYLW02] Gao Cong, Lan Yi, Bing Liu, and Ke Wang. Discovering frequent substructures from hierarchical semi-structured data. In *Proc. of the Second SIAM International Conference on Data Mining*, 2002.

[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.

[GC02] Shayan Ghazizadeh and Sudarshan Chawathe. Discovering freuqent structures using summaries. Technical report, University of Maryland, Computer Science Department, 2002.

[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.

[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.

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

[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.

[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.