Discovering Frequent Graph Patterns Using Disjoint Paths

Ehud Gudes, *Member*, *IEEE Computer Society*,
Solomon Eyal Shimony, *Member*, *IEEE Computer Society*, and Natalia Vanetik

Abstract—Whereas data mining in structured data focuses on frequent data values, in semistructured and graph data mining, the issue is frequent labels and common specific topologies. Here, the structure of the data is just as important as its content. We study the problem of discovering typical patterns of graph data, a task made difficult because of the complexity of required subtasks, especially subgraph isomorphism. In this paper, we propose a new Apriori-based algorithm for mining graph data, where the basic building blocks are relatively large, disjoint paths. The algorithm is proven to be sound and complete. Empirical evidence shows practical advantages of our approach for certain categories of graphs.

Index Terms—Database applications, data mining, mining methods and algorithms, Web mining, graph mining.

1 Introduction

Due to increasing amounts of structured and unstructured data collected by various companies and institutions, the importance of data mining has grown significantly over the last several years. Whereas, in the past, data mining was mainly applied to structured data and flat files, there is growing interest in mining and discovering frequent patterns in semistructured data such as Web data ([23], [35], [2]), chemical compounds data [8], [27], or biological data [29]. The focus of this paper is on discovering such frequent patterns in the form of (possibly labeled) graphs and a new algorithm for this difficult task.

Semistructured data appears when the source does not impose a rigid structure on the data, such as the Web, or when data is combined from several heterogeneous sources. Unlike unstructured raw data (like image and sound), semistructured data does have some structure, but unlike structured data (such as relational or object-oriented databases), semistructured data has no absolute schema or class fixed in advance. For example, in the Internet Movie Database [28], some movies have more actors than others, some fields (e.g., *Award*) are missing for some movies, some actors have birthdays recorded and some do not, etc. As a result, the structure of objects is irregular and a query over the structure is as important as a query over the data. This structural irregularity, however, does not imply that there is no structural similarity among semistructured objects. On the contrary, it is common for semistructured objects describing the same type of information to have similar structures. For example, every movie object has *Title* and *Director* labels, every *Actor* object has a *Name* label, 50 percent of *Actor* objects have a *Nationality* label, etc. This phenomenon is common in other types of semistructured data as well [19].

• *The authors are with the Department of Computer Science, Ben-Gurion University of the Negev, 84105 Beer-Sheva, Israel.*
E-mail: {ehud, shimony, orlovn}@cs.bgu.ac.il.

Manuscript received 1 Feb. 2005; revised 8 Mar. 2006; accepted 30 May 2006; published online 19 Sept. 2006.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-0043-0205.

While, in the field of structured data mining, frequent data *values* and their common appearances are of interest, in mining semistructured data, the focus is on frequent *labels* and common appearances of subsets of such labels (in terms of XML, one would look for frequent occurrences of structures of elements or attributes, disregarding the attribute values). Therefore, frequently, a common model is a *graph*, with labels on nodes, on edges, or on both (the transformation between these types of model is quite simple). In this paper, we assume the model of a directed or undirected graph (or set of graphs) with node labels and our task is to find frequent patterns in such a graph. For example, see Fig. 1, depicting some frequent graph patterns found by our algorithm in an XML movie database [28].

1.1 Graph Mining Applications

Discovering and understanding frequent patterns that represent a sufficiently large part of a semistructured database can be useful in several application areas:

Improving Database Storage and Design [10], [31]. Semistructured data sets (a typical example being XML data), carry their own schema information. Though required for data exchange and integration, such schema incorporation entails considerable space overhead, since the schema information is stored with the data (e.g., element names in XML). Because of this overhead, commercial database management systems often store XML data in relational databases. Semistructured data can always be stored as a ternary relation, since the data is an edge-labeled graph, but this is no better than storing the schema with the data. A "good" mapping (in terms of disk space or fragmentation) from a semistructured data instance into a relational schema is desirable. Frequent patterns discovered in the semistructured data can be used for that purpose since they can help generate the basic relations, while the nonfrequent patterns would be stored as "overflow" relations (see [10]).

Efficient Indexing and Querying. Querying a semistructured database is an important and common activity. Numerous query languages were proposed for this purpose (see [9], [3]). To speed up query processing, several indexing techniques were proposed for semistructured

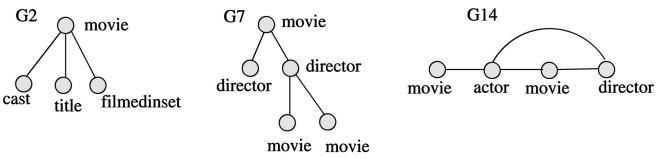

Fig. 1. Pattern examples.

and XML data [13], [25]. Recently, it was realized that full indexing on all possible labels and paths in a semistructured data is not practical. The APEX indexing scheme [5] suggests indexing mainly on frequent paths, where the frequent paths are found by data mining techniques. This idea of using mining for indexing can naturally be generalized to general graphs, as proposed in [31], where paths are used for finding all occurrences of a query graph in the database.

User Preference-Based and User Modeling Applications [11], [38]. An important goal for Web-page design is to provide viewer-oriented personalization of Web-page content. Designers often strive to condition Web-page content and appearance on the current preferences of the viewer and probably on some underlying structure of the Web-page content. In order to optimize such content, one often refers to data mining. When the semistructured database is a collection of user traversal patterns, one can derive expected user behavior from knowledge about frequent traversal patterns of the same user collected over a certain period of time. This results in useful applications, e.g., placing advertisements in proper places and better customer/user classification and behavior analysis. In past work, Web navigation patterns were usually represented as paths or trees and, for this type of problem, tree and path mining are most relevant [4]. However, if one looks at sets of related Web navigation patterns or at behavior over time, one gets more complex patterns which can be represented by graphs, motivating the use of graph mining. Another application related to user behavior is the area of *social networks*, analyzing which is an important field in communication and in security applications [12]. An example of a social network database based on e-mails is used in our study.

1.2 Categories of Graph Mining Problems

In the past, most work done in this field dealt with either single path patterns [2] or treelike patterns [23], [35], [4]. However, much of the data on the web is graphlike, either cyclic or acyclic, motivating the mining of general graph data. The field of graph mining received much attention in recent years and several well-known algorithms were developed, such as AGM [17], FSG [20], gSpan [39], CloseGraph [40], and AdiMine [34]. In this paper, we present a new algorithm for mining frequent patterns in semistructured data, where the data is modeled as a labeled graph. Our algorithm handles general unrestricted graphs, directed or undirected.

There are two distinct problem formulations for frequent pattern mining in graph data sets. In the first, known as the *graph-transaction setting*, the input to the pattern mining algorithm is a set of (usually) relatively small graphs and a pattern is considered frequent if it appears in a large number or fraction of the graphs. Note that a pattern occurrence is counted only once per transaction, independent of possible multiple occurrences in the same transaction. A typical application for this formulation is finding frequent subgraphs in molecular transactions [20].

In the second setting, the frequency of a pattern is based on the number of its occurrences (i.e., embeddings) of a pattern in all the data, counting multiple occurrences per transaction. For this setting, one can assume without loss of generality that the input is a single graph because one can always treat multiple graphs as a single graph with disconnected components. For historical reasons, we refer to this formulation as the *single-graph setting* [21], although neither the problem formulation nor the algorithms are limited in this manner. Due to the inherent differences in characteristics of the problem formulation, algorithms developed for the graph-transaction setting cannot handle the single-graph setting, whereas the latter algorithms can be used to solve the former problem. In recent years, a number of efficient and scalable algorithms have been developed to find patterns in the graph-transaction setting [19], [20], [39], [18], [15], [16], [6]. These algorithms are complete in the sense that they are guaranteed to discover all frequent subgraphs and were shown to scale to very large graph data sets. However, developing algorithms that are capable of finding patterns in cases where each transaction is a large graph, and especially the single-graph setting, has received much less attention, despite the fact that this problem setting is more generic and applicable to a wider range of data sets and application domains than the former problem. Other than our own papers [32], [33], the most recent paper dealing with the single-graph setting is [22], discussed in Section 5.

1.3 Overview of the Proposed Algorithm

The algorithm presented in this paper uses breadth-first enumeration and is based on the Apriori algorithm [1]. These algorithms use an admissibility property (defined below) of the support measure in order to prune candidate patterns without checking their support directly, while ensuring completeness. Since a pattern is considered to be *frequent* in a data set graph if its support measure is greater than a user-provided threshold, then once a pattern has support smaller than the threshold, all of its superpatterns can be pruned or potentially not even be generated in the first place.

Let the *support measure* S be a function from graph patterns and data set graphs to real numbers (usually in [0, 1]). As usually the data set graph is understood, this argument to S is omitted. S obeys the *admissibility* constraint (also called antimonotonicity, or downward closure) if every subgraph of a frequent pattern is also frequent [1], [14]. Formally:

Definition ***(admissible support measure).*** *A support measure S is* admissible *if, for every pattern P, $S(P) \geq 0$ and for all patterns P_1, P_2 such that $P_1 \subseteq P_2$, we have $S(P_1) \geq S(P_2)$.*

Apriori-based algorithms compose candidate patterns from building blocks that vary between algorithms. In our algorithm, the building block is a complete path (see the next section for precise definitions)—as seen in the following (extremely simplified) outline of our algorithm:

1. Find all patterns composed of a single path by directly counting the number of occurrences of these patterns in the data set. Eliminate the nonfrequent patterns.
2. Find all candidate patterns composed of two frequent paths and eliminate the nonfrequent patterns.

3. At each successive step n:
   a. Construct candidate patterns from smaller frequent ones that have a common "core." Specifically, generate patterns with $n+1$ paths by merging two patterns with n paths that have a common core with $n-1$ paths. A simple example for $n=2$ is shown in Fig. 8: Two graphs, each consisting of two paths, with an identical core consisting of one path, are merged to create a graph with three paths. In general, this construction, the heart of the algorithm, is quite complex.
   b. Prune candidates that are not frequent.
   c. Stop when no more frequent patterns can be generated.

The above outline is precisely the same as for most Apriori-based algorithms, the crucial difference being that while for itemset mining, the building blocks are items, and for most graph mining algorithms (such as FSG or gSpan), the building blocks are edges or nodes, in our algorithm, the building blocks are the (typically much larger) *edge-disjoint paths*. Making the building block larger allows for a smaller number of iterations, as well as for a smaller number of candidate patterns that need to be tested for support—the main goal of our scheme. Since testing support of a pattern is expensive, especially for graph data, it is important to improve pruning, even if it entails considerable overhead over the naive methods of generating and testing patterns. Another complication is that achieving completeness becomes nontrivial, and considerable space is devoted in this paper to how completeness can be provably maintained.

1.4 Contributions and Outline of the Paper

The idea of using paths as building blocks in a graph mining algorithm was presented briefly in an earlier, conference version of this paper [32]. The operators used to define graph composition, which allow for efficient implementation of the graph merge in practice, are a new contribution of this version. A major issue in this respect is proving that our edge-disjoint path-based algorithm is complete. This proof of correctness (Section 3) is a previously unpublished, nontrivial main contribution of this paper.

In attempting to find frequently occurring subgraph patterns within a graph, computing the frequency of occurrence of the pattern in the larger graph (the database) and the support measure is an intensive computational step. This involves multiple computations of the subgraph isomorphism problem, which is a hard problem. In order to decrease the number of extremely expensive support computations, we must discard, as early as possible, as many candidate patterns as possible. This is a general property of our algorithm. Minimizing the number of expensive support computations is the second major contribution of this paper. This advantage is more prominent when the transaction graphs are large, and even more so in the "single-graph" setting, where the support computation tends to be extremely hard.

To prove the feasibility of our scheme, we implemented the proposed algorithms, tested them on some XML databases and synthetic graphs, and compared them to other approaches for counting graphs patterns, mainly the naive and the FSG algorithms. Note that the algorithm presented here is orthogonal to the support measure and therefore can be used for both cases and is compared experimentally to FSG in both cases. The results show that, while in the transaction setting the two algorithms are comparable, in the "single-graph" setting our algorithm shows a significant reduction in the number of candidates generated and therefore in the number of support computations. In our experiments, we dealt with medium-size graph databases (up to 20,000 nodes) since for larger sizes the single-graph case support computation was too heavy computationally for both the FSG algorithm and ours. The experimental evaluation of our algorithm (Section 4) is the third contribution of this paper.

The rest of this paper is organized as follows: We begin with revisiting some graph theoretic notation and results (Section 2), followed by a formal definition of our graph-mining problem and new definitions used in specifying the algorithm. The graph mining algorithm and its correctness proof are then presented (Section 3). Empirical evaluation of our algorithm on both synthetic and real data is examined in Section 4. Section 5 discusses related work, as well as the applicability of our algorithm to other settings.

2 Preliminaries

We begin with revisiting standard terms from the literature. A formal statement of our graph-mining problem is made, followed by a definition of *composition operators* essential for generating candidate graphs in the algorithm. Important basic properties of the operators are stated and proved.

2.1 Paths and Path Covers

We begin by revisiting some graph-theoretic terms and properties. Notation needed later on is also introduced.

A *path* is an alternating sequence of nodes and (their incident) edges that begins and ends with a node and that does not contain any edge more than once. For directed graphs, we require a path to respect the direction of the edges, resulting in a *directed path*. A set P of edge-disjoint paths covering all edges of a graph G exactly once is called a *path cover* of G. A path cover P is called minimal if it has the smallest cardinality of all path covers of G. Clearly, in general, the minimal cover is not unique. The *path number* $p(G)$ is the cardinality of any minimal path cover of G.

In this paper, we use paths as the building blocks in order to create larger graphs, but we are not concerned about how to traverse the paths once they have been created. Henceforth, we ignore the ordering inherent to the path definition and represent a path simply as the set of nodes and edges in the path, i.e., as a graph. Two different paths that have the same set of nodes and edges are thus indistinguishable in our method. Note that we still require that such a graph be traversable as a single path, even though the traversal does not have to be unique.

Removing path P from graph G, denoted by $G \setminus P$, consists of removing all edges of P from G, followed by removing all stand-alone nodes. To compute the path number, we rely on well-known facts:

1. A connected undirected graph $G=(V,E)$ is *Eulerian* (can be covered by a single cyclic path) iff for every $v \in V$, $d(v)$ is even. A connected digraph $G=(V,E)$ is *Eulerian* (can be covered by a single directed cyclic path) iff for every $v \in V$, $d^+(v)=d^-(v)$. (Throughout, we denote the in-degree of v by $d^+(v)$, and the out-degree by $d^-(v)$.)

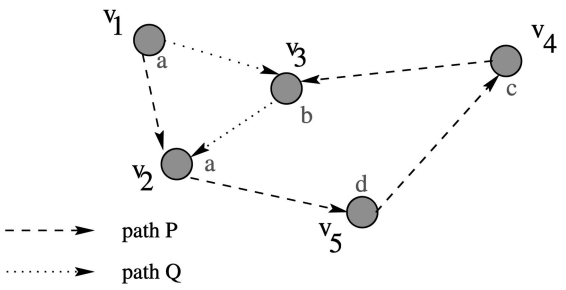

Fig. 2. Representing tuples.

2. For every connected undirected graph $G = (V, E)$, $p(G) = 1$ if G is Eulerian, and $p(G) = |\{v \mid v \in V, d(v)\ is\ odd\}|/2$ otherwise. For every connected directed graph $G = (V, E)$, $p(G) = 1$ if G is Eulerian and $p(G) = (\sum_{v \in V} |d^+(v) - d^-(v)|)/2$ otherwise.

Observe that the path number of a graph is never greater than the number of edges, being in fact much smaller in most cases—especially for undirected graphs. Thus, paths as building blocks should decrease the number of iterations in the algorithm, as well as improve the pruning.

2.2 Problem Statement

A *labeled graph* is a graph that has a label associated with each node v, denoted by $label(v)$. We assume without loss of generality that the data set (as well as the pattern) graph is labeled (otherwise, assign to all nodes in the graph the same arbitrary label). Given two graphs $G' = (V', E')$ and $G'' = (V'', E'')$, a *label-preserving isomorphism* between G' and G'' is a graph isomorphism $\phi : V' \to V''$ such that, for every $v \in V'$, $label(v) = label(\phi(v))$. When such an isomorphism exists, denote by $G' \approx G''$ the fact that the graphs are isomorphic. P is a *graph pattern* in graph G if it is isomorphic to a connected subgraph of G.

Our problem is formally defined as follows. Given a data set labeled graph G, a support measure S over pattern graphs, and a support threshold σ, find all pattern graphs P with support $S(P) \geq \sigma$ in G. Recall that the input can be a set of graphs as well as a single graph without loss of generality throughout.

2.3 Lexicographic Ordering

To facilitate efficient indexing of path covers in a graph, we use a canonical representation of paths and path sequences. The lexicographical ordering over paths uses node labels and degrees of nodes in paths, as follows:

A path P uniquely defines the graph $(V(P), E(P))$: the nodes traversed by the path, and the edges traversed by the path, respectively. For node $v \in V(P)$, the *path degrees* $d^+_P(v)$ and $d^-_P(v)$ are the in-degree and out degree, respectively, of v in $(V(P), E(P))$. For undirected paths, the *path degree* $d_P(v)$ of v is simply the degree of v in $(V(P), E(P))$.

Let P be a directed path and let $v \in V(P)$. A node in a path is represented by a representing tuple (see Fig. 2), defined as follows:

$$RT_P(v) := (label(v), d^+_P(v), d^-_P(v)).$$

For undirected paths, the representing tuple is likewise defined as

$$RT_P(v) := (label(v), d_P(v)).$$

Assuming a natural complete ordering between labels, as well as the natural complete ordering between integers, a lexicographical ordering between the node representation tuples of u and v, denoted $v \prec_L u$, is understood. Likewise, equality operator $v =_L u$ denotes equality of the respective representing tuples.

Paths are indexed by a *path descriptor*, defined as follows: Given a path P, sort $V(P)$ using the order $\prec_L$. The resulting sorted sequence, denoted by $de(P)$, is the path descriptor of P. The order $\prec_p$ between paths is a lexicographic ordering between path descriptors, using $\prec_L$ for elementwise comparison. When the path descriptors of P and Q are lexicographically equal (which occurs just when the sequences are equal), we write $P =_p Q$. If $de(P)$ and $de(Q)$ are sequences of unequal length, such that the shorter sequence (let it be $de(P)$ without loss of generality) is a prefix of the longer sequence, we will use the convention that, in this case, $P \prec_p Q$. Note that $P = Q$ entails $P =_p Q$, but not vice versa.

Fig. 2 shows a path cover of size 2 of a graph, where the paths are $P = v_1, v_2, v_5, v_4, v_3$ and $Q = v_1, v_3, v_2$. The path descriptors are

$$de(P) = (a, 0, 1), (a, 1, 1), (b, 1, 0), (c, 1, 1), (d, 1, 1),$$

and

$$de(Q) = (a, 0, 1), (a, 1, 0), (b, 1, 1).$$

Since $de(Q) \prec_{lex} de(P)$ (because $(a, 0, 1) =_{lex} (a, 0, 1)$ and $(a, 1, 0) \prec_{lex} (a, 1, 1)$), we have $Q \prec_p P$.

Observation 1. *$\prec_p$ is transitive and complete (that is, every pair of paths is comparable).*

Finally, multisets of paths (which are used to represent a decomposition of a graph into paths) are indexed by composition descriptors, defined as follows: Let $\mathcal{P}$ be a multiset of paths. The decomposition descriptor of $\mathcal{P}$, denoted $dc(\mathcal{P})$, is the sorted sequence of the elements of the multiset $\{de(P) | P \in \mathcal{P}\}$, sorted according to the ordering $\prec_p$. The ordering $\prec_{lex}$ over multisets of paths is defined as a lexicographic ordering of their composition descriptors.

A minimal path cover $\mathcal{P}$ of a graph G is called $P - minimal$ if there is no minimal path cover $\mathcal{Q}$ for which $\mathcal{Q} \prec_{lex} \mathcal{P}$. Observe that there may be more than one $P - minimal$ path cover for a given graph G, but the composition descriptors of all the minimal path covers of G are equal.

2.4 Properties of Path Covers

In our data mining algorithm, we intend to keep in the nth frequent candidate set only graphs with path number n. The path number of a graph can be computed in linear time, as it can be determined uniquely from the multiset of node degrees. In order to correctly produce candidates with path number $(n + 1)$ by combining graph pairs with path number n, several basic properties of the path covers must be shown to guarantee *completeness* of our algorithm:

1. Removing a path (in a minimal path cover) from a graph reduces the path number by 1.
2. For every connected graph G and minimal path cover of size $n > 2$, there are at least two paths in the cover, each of which can be subtracted from G, leaving the resulting graph connected.
3. If path number is greater than 1, all paths in a minimal cover are noncyclic.

These properties are stated and proved below.

Theorem 1. *Let G be a graph (directed or undirected) with path number $n > 1$, and let $\mathcal{P}$ be a minimal path cover of G. Then, for every path $P \in \mathcal{P}$, the graph $G' = G \setminus P$ has path number $n - 1$.*

Proof. Clearly, $\mathcal{P} \setminus P$ is a path cover of $G \setminus P$ and, thus, $p(G \setminus P) \leq n - 1$. Now, let $\mathcal{P}'$ be a path cover of $G \setminus P$ of size $n' < n - 1$. Then, $\mathcal{P}' \cup \{P\}$ is a path cover of G of size $n' + 1 < n$, a contradiction. □

Theorem 2. *Let $G = (V, E)$ be a connected graph with $p(G) = n \geq 2$ and let $(P_1, \ldots, P_n)$ be a minimal path decomposition (assuming any arbitrary ordering on the paths). Then, there exist $1 \leq i < j \leq n$ such that graphs $G \setminus P_i$ and $G \setminus P_j$ are connected.*

Proof. Define the undirected "decomposition graph" $G' = (V', E')$ of the decomposition, as follows:

$$V' = \{v_i | 1 \leq i \leq n\},$$

and $\{v_i, v_j\} \in E'$ just when P_i, P_j have at least one node in common. Clearly G is connected if and only if G' is connected. This property also holds for any $G \setminus P_i$ and its corresponding decomposition graph, where the latter decomposition graph is equal to G' with node v_i and its incident edges removed. Since G is connected, so is the decomposition graph G'. It is well known that every connected graph with more than two nodes has at least two nodes, each of which can be removed (together with their incident edges), leaving the graph connected. Let v_i, v_j be two such nodes in G' (with $i \neq j$). By construction, this implies that $G \setminus P_i$ and $G \setminus P_j$ are both connected graphs. □

Finally, a minimal path cover of a connected graph with a path number greater than 1 consists only of noncyclic paths, i.e., paths whose start and end vertices are different. That is because any cycle P can be at any point v where it intersects another path Q and merges into path Q—thereby reducing the size of the cover (contradicting the minimality of the path cover). Thus, we can construct all graphs with path number $n < 1$ just from noncyclic frequent paths. For undirected graphs, we also can show this:

Lemma 1. *Let $G = (V, E)$ be an undirected graph with minimal path cover $\mathcal{P}$, with $p(G) \geq 2$. Then, every path $P \in \mathcal{P}$ starts at a node v of odd degree and ends at a node u of odd degree, and $v \neq u$.*

Proof. From the above result, all paths in the path covers are noncyclic. Let $P \in \mathcal{P}$ and let node v be the start of P (alternately, P ends at v, but not both), implying that P has an odd number of edges incident on v. Then, for all $Q \in \mathcal{P}$, path $Q \neq P$ contains an even number of edges incident to v (because, otherwise, Q either starts or ends at v and can be merged with P into a single path, again contradicting minimality of $\mathcal{P}$). The degree of v is the sum of the number of edges incident on v over all paths in the cover, which (being the sum of even numbers plus exactly one odd number) is odd. □

2.5 Compositions and Graph Merging—Notation and Definitions

In this section, we define the basic operations used to combine graphs with a common core, preceded by some required notation.

TABLE 1
Composition Tuple-Set τ on P_1, P_2, P_3

Node	P_1	P_2	P_3
v_1	a_1	$\perp$	$\perp$
v_2	a_2	b_2	$\perp$
v_3	a_3	$\perp$	$\perp$
v_4	$\perp$	b_1	$\perp$
v_5	$\perp$	b_3	c_3
v_6	$\perp$	$\perp$	c_1
v_7	$\perp$	$\perp$	c_2

2.5.1 Notation

For sequences and tuples, we use the following standard notation. Let t be a sequence of length n (or n-tuple). Then, for $1 \leq i \leq j \leq n$ we denote the ith element of t by $t[i]$, and $t[i : j]$ denotes the subsequence (subtuple) of t starting at i and ending at j, inclusive. The above subscripting and subsequence operators are also applied to sets of tuples. Thus, if T is a set of tuples, then $T[i : j] = \{t[i : j] | t \in T\}$. A set subtraction operator inside the square brackets indicates removal of the subtracted elements from the tuple (respectively, set of tuples). Thus, $t[1 : n \setminus j]$ indicates an (n-1)-tuple consisting of all the elements of t except $t[j]$, in the same order as in t.

We use the *dot* operator as a sequence (respectively, tuple) concatenation operator. Applied to a simple element, we mean concatenation with the respective 1-tuple. For example, when e is a simple element, $t.e$ denotes an (n+1)-tuple, with $(t.e)[1 : n] = t$, and $(t.e)[n + 1] = e$. When referring to graph elements, we use $\perp$ to denote a null element. By $t = \perp$, we mean that in tuple t all elements are equal to $\perp$. We define a composition operator (denoted as $+$) between graph elements as follows: Let a be a non-null graph element. The operator is defined as follows:

$$a + a = a + \perp = \perp + a = a$$
$$\perp + \perp = \perp.$$

The composition of two different, non-null elements is undefined (as used in this paper, such a composition is called inconsistent). The composition operator is also applied as a vector operator, to pairs of n-tuples, denoting elementwise composition, thus:

$$(a, \perp, b, \perp) + (a, b, \perp, \perp) = (a, b, b, \perp).$$

A vector composition where any of the elementwise compositions is undefined is also undefined (inconsistent).

2.5.2 Graph Compositions

As one of the steps of our algorithm, two graphs (each composed of a set of paths) are merged to create a larger graph. In order to facilitate operations on such composite graphs, we define the notion of *composition tuple-set* (a *composition* for short). See Table 1 for an example.

Definition 1. *Let $\mathcal{G}$ be a set of graphs. A composition tuple-set τ of width n over $\mathcal{G}$ is a pair $(G(\tau), tuples(\tau))$, where $G(\tau)$ is an n-tuple with each element designating a graph in $\mathcal{G}$, and $T = tuples(\tau)$ is a set of n-tuples, where, for every tuple $t \in T$ and every $1 \leq i \leq n$, the element $t[i]$ designates either a node in the graph $G(T)[i]$ or $\perp$. A tuple t is* label-consistent *if, for all*

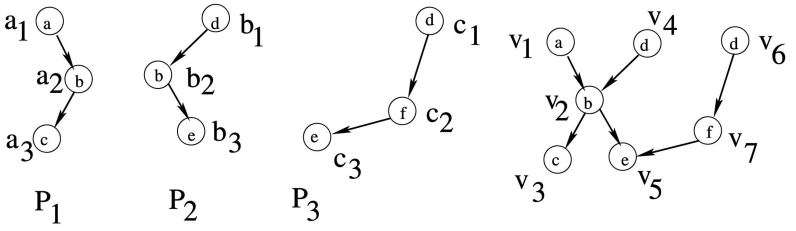

Fig. 3. Graph G composed from three edge-disjoint paths: P_1, P_2, P_3.

$1 \leq i, j \leq n$ *for which* $t[i]$ *and* $t[j]$ *are non-null, the nodes designated by* $t[i]$ *and* $t[j]$ *have the same label.*

The number of elements in each tuple in $tuples(\tau)$ will be denoted by $width(\tau)$. Observe that $G(\tau)$ may have more than one element referring to the same graph. In our algorithm, the set $\mathcal{G}$ will always contain single paths, i.e., graphs that have a path number of 1, but the notation can also be used for composition of other types of graph. We will henceforth assume that $\mathcal{G}$ is the set of paths in our data set graph and thus omit reference to $\mathcal{G}$. (In practice, we actually take this set to be the set of just the *frequent* paths, for reasons of efficiency.) The semantics of a composition is a (composite) graph, called the *induced graph*, which has one node for every tuple in T. In order to define the induced graph, we first wish to make sure that the composition tuple-set defines an edge-disjoint composition of subgraphs that does not distort the subgraphs of which it consists.

Definition 2. *A composition* τ *(over* $\mathcal{G}$*) is* consistent *if all the following conditions hold:*

1. *For every* $1 \leq i \leq width(T)$ *and every node*

$$v \in V(G(T)[i]).$$

there exists a unique $t \in T$ *such that* $t[i] = v$*. (The* node consistency *condition: There is a unique representing tuple for every node.)*
2. *Every* $t \in T$ *is non-null and label-consistent.*
3. *For every pair of tuples* $t_1, t_2 \in T$*, we have*

$$|\{i \mid (t_1[i], t_2[i]) \in E(G(T)[i])\}| \leq 1.$$

(The edge disjointness *condition: Each pair of (induced) vertices has an edge in at most one of the graphs participating in* T*.)*

Two composition tuple sets are equivalent if they are equal or one is equal to the other under a permutation of the indices. (By "under a permutation," we mean any arbitrary permutation, but with the *same permutation* applied to all the tuples in $tuples(\tau)$ and to $G(\tau)$.) The graph induced by a composition tuple-set $\tau = (G(\tau), T)$ is denoted by $\Omega(\tau)$ and defined as follows:

Definition 3. $\Omega(\tau) = (V, E)$*, with* $V = \{\nu(t) | t \in T\}$ *(where* ν *is an arbitrary function that assigns a unique node to every tuple* t*), and*

$$E = \{(\nu(t_1), \nu(t_2)) | t_1, t_2 \in T \wedge \exists i (t_1[i], t_2[i]) \in E(G(\tau)[i])\}.$$

That is, the induced graph has a node for every tuple in T and an edge between a pair of nodes just when one of the subgraphs composing τ has an edge between these nodes. Observe that the edge disjointness condition ensures that this subgraph is unique. When used to compose new graphs, the function ν evaluates to a *new* unique node, i.e., one that does not appear elsewhere in the system.

Fig. 3 shows a graph consisting of three paths, P_1, P_2, P_3, and Table 1 presents a corresponding composition tuple-set, i.e., the graph is an induced graph of the tuple-set.

Observation 2. *Let graph* G *be covered by* n *mutually edge-disjoint subgraphs* $G_1, G_2, \ldots, G_n$*. Then,* G *is identical to the graph induced by the composition tuple-set* $\tau = (G(T), T)$ *constructed as follows:* $G(T) = (G_1, G_2, \ldots, G_n)$*, and* $tuples(T)$ *consists of* $|V(G)|$ *tuples, one unique tuple* t *for each node in* $v \in G$*. Denote the bijection from tuples to nodes by* μ *and let* $t[j] = \mu(t)$ *if* $\mu(t) \in V(G(T)[j])$ *and, otherwise,* $t[j] = \perp$*. A composition tuple-set defined as above is called a* natural *composition tuple-set with regard to* G *and its cover.*

Proposition 1. *The composition tuple-set* τ *is consistent and* $\Omega(T)$ *is isomorphic to* G *under the "natural" isomorphism, where* $\nu(t) \approx \mu(t)$ *for all* $t \in T$*.*

Proof. Observe that T obeys the node-consistency condition by construction. Since the graph cover of G is edge-disjoint, an edge in G implies an edge in exactly one of the subgraphs and, thus, T observes the edge-disjointness condition. Clearly $\nu(t) \approx \mu(t)$ as defined above is an isomorphism between $\Omega(T)$ and G, by construction. □

Until this point, we did not constrain the type of subgraphs $G(\tau)$ in a composition. Henceforth, we will assume that all these subgraphs have a path cover of size 1, i.e., each such subgraph is a single path. Finally, we introduce the notion of P-minimal compositions; as an extension of this notion in path covers, a composition tuple-set τ is P-minimal if there is no τ' such that $\Omega(\tau') = \Omega(\tau)$ and $G(\tau') \prec_{lex} G(\tau)$.

2.5.3 Operators on Compositions

We proceed to define operators on composition tuple-sets, and the respective operations on the induced graph. The first desired operation is a projection operator—keeping only certain parts of all tuples (corresponding to keeping only some parts of the induced graph). This operation uses our previously defined index range notation. Thus, by

$$\tau' = \tau[i:j](= (G(\tau)[i:j], tuples(\tau)[i:j] \setminus \perp)),$$

we indicate that τ' is a projection of the composition τ onto columns i to j inclusive. Observe that removing some elements of a non-null tuple may result in a null tuple and that such tuples are dropped by the projection operation. Likewise, to indicate removal of subgraph i from a composition T of width n:

$$T' = T[1:n \setminus i]$$
$$(= (G(T)[(1:n \setminus i], tuples(T)[1:n \setminus i]) \setminus \perp).$$

The resulting τ' is a composition of width $n-1$.

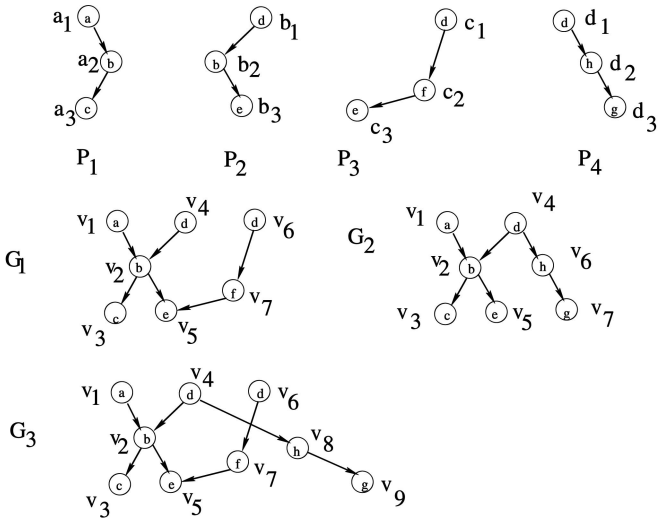

Fig. 4. Induced graph of a bijective sum.

In general, projection operations can cause projected tuples to become equal, thus reducing the number of tuples in the resulting composition. However, for consistent compositions, this can occur only for tuples which then become null and are dropped in the projection. This is due to the following property, which follows immediately from the node consistency condition:

Proposition 2. *Let $R \in [1,n]$ be an arbitrary sequence of indices, τ a consistent composition, and $t_1, t_2 \in tuples(\tau)$ with $t_1 \neq t_2$. Then, $t_1[R] = t_2[R]$ implies $t_1[R] = t_2[R] = \perp$.*

Having defined the required notation, the main operators used in our algorithm are defined below. Creating a larger graph from two smaller graphs is done using the bijective sum operator, defined as follows:

Definition 4 (Bijective Sum). *Let $\tau_1 = \tau_2$ be compositions, each of width $n-1$, such that*

$$\tau_1[1:(n-2)] = \tau_2[1:(n-2)].$$

Let $T_1 = tuples(\tau_1)$ and $T_2 = tuples(\tau_2)$. The bijective sum of τ_1 and τ_2, denoted $BS(\tau_1, \tau_2)$, is a composition τ of width n with $G(\tau) = G(\tau_1).G(\tau_2)[n-1]$ and with $tuples(\tau)$ being (the union of) the following sets of tuples:

1.
$$\{t_1.t_2[n-1] \mid t_1 \in T_1, t_2 \in T_2, \\ t_1[1:(n-2)] = t_2[1:(n-2)] \neq \perp\}.$$

2.
$$\{\perp^{n-2}.t[n-1].\perp \mid t \in T_1, t[1:(n-2)] = \perp\}$$

(where $\perp^i$ means an all-$\perp$ i-tuple).

3.
$$\{\perp^{n-2}.\perp.t[n-1] \mid t \in T_2, t[1:(n-2)] = \perp\}.$$

The intuition for this definition is as follows, by considering the induced graphs of the composition tuple-sets (see Fig. 4). Now, map (and consider as the same node) the nodes in the induced graphs standing for the tuples that include $T_1[1:(n-2)]$ to those induced by $T_2[1:(n-2)]$, basing the mapping on tuple equality. Tuples in (1) correspond to nodes appearing in the induced graphs of both τ_1 and τ_2. Tuples in (2) correspond to nodes that appear in the graph induced by τ_1, but do not appear in τ_2. Likewise, tuples in (3) correspond to nodes that appear in the graph induced by τ_2, but do not appear in τ_1. Henceforth, the construction (1) above will be called *type 1 construction* and the respective generated tuples are called *type 1 tuples*. Likewise for items (2) and (3) above. Observe that in some cases the result of a bijective sum may be inconsistent due to a violation of the edge disjointness condition. Our algorithm will discard the results of such inconsistent bijective sums.

TABLE 2
Bijective Sum

T_1				T_2			
Node	P_1	P_2	P_3	Node	P_1	P_2	P_4
v_1	a_1			u_1	a_1		
v_2	a_2	b_2		u_2	a_2	b_2	
v_3	a_3			u_3	a_3		
v_4		b_1		u_4		b_1	d_1
v_5		b_3	c_3	u_5		b_3	
v_6			c_1	u_6			d_2
v_7			c_2	u_7			d_3

T_3				
Node	P_1	P_2	P_3	P_4
w_1	a_1			
w_2	a_2	b_2		
w_3	a_3			
w_4		b_1		d_1
w_5		b_3	c_3	
w_6			c_1	
w_7			c_2	
w_8				d_2
w_9				d_3

The definition of bijective sum can easily be generalized to allow for the equivalent part of τ_1 and τ_2 to be any subset of indices of size $n-2$, not necessarily $[1:(n-2)]$ and not necessarily in sorted order. However, this would make the notation exceedingly cumbersome. Equivalently, one can view this generalized definition as permuting the element positions of τ_1 and τ_2 in order to get $\tau_1[1:(n-2)] = \tau_2[1:(n-2)]$, performing the bijective sum, and arbitrarily permuting the positions of τ. In the description of the algorithm, we use this permutation scheme in order to simplify the notation.

Table 2 demonstrates a bijective sum $T_3 = BS(T_1, T_2)$ of two composition tables T_1 and T_2, and in Fig. 4 the respective induced graphs $G_1 = \Omega(T_1), G_2 = \Omega(T_2)$ and $G_3 = \Omega(T_3)$. Null values are shown as blanks.

Observe that lifting the restriction that the width of the composition sets be equal results in a meaningful (as far as the induced graph is concerned) and potentially useful operator. But since our algorithm does not use such a generalization, we shall not discuss this issue further.

Our algorithm also requires an operator that allows nodes induced by tuples of types (2) to be merged with nodes induced by tuples of type (3) after a bijective sum. The merged nodes are determined by a composition of width 2. For this purpose, we define the *splice* operation, as follows (refer to Fig. 5 as an example).

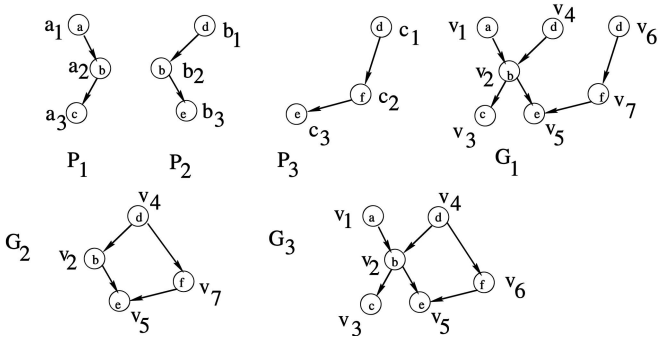

Fig. 5. Induced graph of a splice.

Definition 5 (Splice). *Let τ be a composition of width $n \geq 3$, and S be a composition of width 2, with $G(S) = G(\tau)[(n-1):n]$. The result of* splicing *τ by S, denoted $Splice(\tau, S)$, is a composition τ' with $G(\tau') = G(\tau)$, and $T' = tuples(\tau')$ defined as follows: Denote $T = tuples(\tau), \mathbf{s} = tuples(S)$, and let M be a set of "merged" tuples:*

$$\begin{aligned} M = \{ \quad & \mid t_1 + t_2 \mid t_1, t_2 \in T, \exists s \in \mathbf{s} \\ (\quad & t_1[n-1] = s[1] \neq \perp \wedge t_2[n] = s[2] \neq \perp \\ \wedge \quad & t_1[n] + s[2] = s[2] \wedge t_2[n-1] + s[1] = s[1]) \}. \end{aligned} \tag{1}$$

Let M' be the set of all tuples t_1, t_2 from T being merged above (i.e., that participate in the sum $t_1 + t_2$ in the above definition of M). The tuples in the resulting composition are $T' = M \cup T \setminus M'$.

Observe that $t_1 = t_2$ is allowed in (1). Also, note that it is possible to have S and T such that some of the $t_1 + t_2$ are undefined. In this case, the splice operation is undefined (inconsistent). For example, Table 3 describes composition tuple-sets T_1, T_2 and their splice $T_3 = Splice(T_1, T_2)$. Fig. 5 shows the corresponding induced graphs $G_1 = \Omega(T_1)$, $G_2 = \Omega(T_2)$ and $G_3 = \Omega(T_3)$. In this figure, the paths P_2 and P_3 in G_1 are spliced using information on nodes common to these paths in G_2.

3 The Graph Mining Algorithm

This section presents our algorithm pseudocode for mining frequent graph patterns, which works for both directed and undirected graphs. A proof of correctness and a partial complexity analysis are then developed.

TABLE 3
Splice

T_1				T_2			T_3			
Node	P_1	P_2	P_3	Node	P_2	P_3	Node	P_1	P_2	P_3
v_1	a_1			u_2	b_1	c_1	w_1	a_1		
v_2	a_2	b_2		u_4	b_2		w_2	a_2	b_2	
v_3	a_3			u_5	b_3	c_3	w_3	a_3		
v_4		b_1		u_6		c_2	w_4		b_1	c_1
v_5		b_3	c_3				w_5		b_3	c_3
v_6			c_1				w_6			c_2
v_7			c_2							

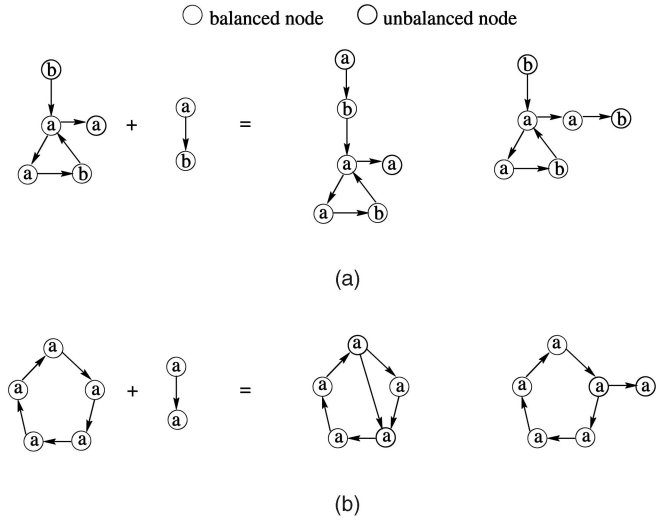

Fig. 6. Phase 1 example. (a) Adding edge to a noncyclic path. (b) Adding edge to a cyclic path.

3.1 Description of the Algorithm

The algorithm consists of three phases. In phase 1, we find all frequent paths (including paths with cycles), starting with frequent nodes and frequent edges. In phase 2, we find all graphs composed of two paths, in other words, we find all possible intersections between pairs of paths from phase 1. In phase 3, we merge pairs of frequent graphs, each consisting of $n-1$ paths, such that the graphs have a common core of $n-2$ paths in an attempt to produce graphs with n paths. Throughout, we assume that some admissible support measure is used. In phases 1 and 3, we construct frequent graph patterns recursively, using the Apriori approach [1].

Phase 1 (see Algorithm 1) constructs the frequent paths considering all frequent paths found in the previous iteration, and potentially adding a frequent edge. Adding the edges is done using the ExpandPath function. First, consider the case for directed graphs in ExpandPath, which considers adding an outgoing edge from some nodes in the path. If the path is cyclic (not necessarily a simple cycle) we can add the outgoing edge anywhere, provided the node labels match (see Fig. 6b for examples). Otherwise, we can only add an outgoing edge at a node that has an in-degree greater than the out-degree—there can be only one such node if P is a path (Fig. 6a). We use the node set X to denote the nodes where an edge can be added. There are now two cases: adding an additional node to the path (step 1, and see Fig. 6 (a and b2) for an example), and adding an edge to a node already on the path (step 2, see Fig. 6b1 for an example) In the graph, one could add an edge at any node that has unequal in-degree and out-degree (an unbalanced node), but it is sufficient to add just the outgoing edge, as shown in the proof of correctness later on.

Algorithm 1 Frequent paths—Phase 1.

Notation: F_i is a set containing frequent graph patterns that are paths with i edges;
$\psi(.)$ is a function that creates a new node with the same label as its argument.
C_i is a candidate set for 1-path patterns with i edges.
Output: L_1, a sorted set containing the frequent paths.

1. Find all frequent nodes and add them to F_0.
2. Find and add to F_1 all frequent edges by scanning the data set and set $k := 2$.
3. Set $C_k := \emptyset$, $F_k := \emptyset$.
4. For every path $P = (V, E) \in F_{k-1}$ and for every $e \in F_1$ do: $C_k := C_k \cup \text{ExpandPath}(P, e)$.
5. For every $G \in C_k$, add G to F_k if G is frequent, and F_k contains no graph isomorphic to G.
6. If $F_k \neq \emptyset$ set $k := k + 1$ and goto step 3.
7. Output $L_1 = \bigcup_{i=1}^{k-1} F_i$ sorted according to $\prec_p$.

Function ExpandPath(P, e) for **directed graphs**
Let *Result* $= \emptyset$. Denote e by (v, u).
If there is a node $x \in V$ s.t.
$d^-(x) < d^+(x)$, let $X = \{x\}$.
Otherwise (i.e., P is cyclic), let $X = V$.

1. For every $x \in X$ s.t. *label*$(x) =$ *label*(v) add $G = (V \cup \{\psi(u)\}, E \cup \{(x, \psi(u))\})$ to *Result*.
2. For every $x \in X$ s.t. *label*$(x) =$ *label*(v), and every $y \in V \setminus x$ s.t. *label*$(y) =$ *label*(u) and $(x, y) \notin E$, add graph $G = (V, E \cup \{(x, y)\})$ to *Result*.

Function ExpandPath (P, e) for **undirected graphs**
Let *Result* $= \emptyset$. Denote e by $\{v, u\}$.
Let X be the set of nodes of odd degree in P. If X is empty, (i.e., P is cyclic), let $X = V$.

1. For every $x \in X$ s.t. *label*$(x) =$ *label*(v), add $G = (V \cup \{\psi(u)\}, E \cup \{\{x, \psi(u)\}\})$ to *Result*.
2. For every $x \in X$, s.t. *label*$(x) =$ *label*(u) add $G = (V \cup \{\psi(v)\}, E \cup \{\{x, \psi(v)\}\})$ to *Result*.
3. For every $x, y \in V$, $x \neq y$ s.t. $\{x, y\} \notin E$, *label*$(x) =$ *label*(v), *label*$(y) =$ *label*(u) s.t. at least one of x, y is in X, add $G = (V, E \cup \{\{x, y\}\})$ to *Result*.

The treatment of undirected graphs is practically the same, differing only in that ExpandPath for undirected graphs considers adding an undirected edge. Here, an edge can be added anywhere if the path is cyclic or at one of the two odd-degree nodes if the path is noncyclic. When adding an additional node (steps 1 and 2 in Algorithm 1, ExpandPath for undirected graphs) the new node can be at either end of the edge. Observe that only in phase I does there exist a significant difference between directed and undirected graphs, except for code hidden in computing the number of paths (which is a simple counting of node degrees) and in the support measure (which is external and largely independent of our algorithm).

Phase 2 (see Algorithm 2) constructs the frequent graphs with path number 2, by combining one-path graphs. The nontrivial steps are steps 2, 3, and 4, where, in step 2, *all* possible compositions of the two paths are considered and, in step 4, both the path number and the support measure are calculated; in step 3, all non-P-minimal isomorphic graphs are removed. Fig. 7 shows (in terms of the lexicographic order we defined earlier) how several different 2-path graphs are constructed from two paths.

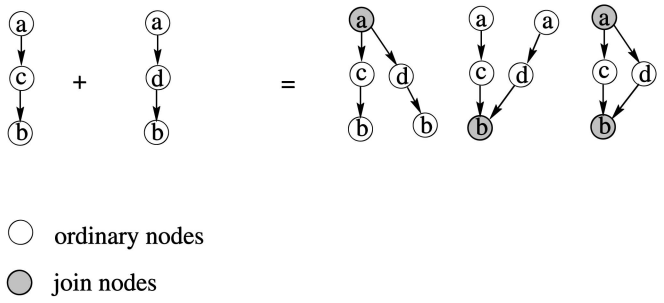

Fig. 7. Phase 2 example.

Algorithm 2 Frequent path pairs—Phase 2

Notation: L_2 is a set that contains composition tuple-sets of frequent graph patterns with\path number 2.
C_2 is a candidate set for the above composition tuple-sets.

1. Let $C_2 = \emptyset$, $L_2 = \emptyset$.
2. For every pair of paths $P_1, P_2 \in L_1$ and every consistent composition tuple-set τ with $G(\tau) = (P_1, P_2)$, s.t. $\Omega(\tau)$ is connected and $p(\Omega(\tau)) = 2$, add tuple-set τ to C_2.
3. Remove from C_2 all tuple-sets that are not P-minimal.
4. For every tuple-set $\tau \in C_2$, if $\Omega(\tau)$ is frequent, add τ to L_2.
5. Output graphs $\{\Omega(\tau) \mid \tau \in L_2\}$.

Phase 3 (see Algorithm 3) constructs the frequent graphs with path number n from graphs with path number $n - 1$. The nontrivial step is step 2. In case 2a, the graph is constructed by finding the common $n - 1$ subgraph structure and adding the remaining two paths P_1, P_2 (one from each graph), using the bijective-sum operation. Note that the specification of an "arbitrary permutation" is just a notational convenience, and is not actually implemented this way (it would require an exponential number of tests). Instead, the composition tuple-sets τ_i are represented in sorted order of the paths in $G(\tau_i)$, where each path is represented by its index in the sorted L_1. To check whether two compositions can undergo bijective sum, simply compare the strings of sorted indices of paths in $G(\tau 1)$, $G(\tau 2)$, allowing for up to one substitution, which can be done very efficiently. The number of cases meeting this requirement is typically many orders of magnitude smaller than the number of possible permutations, which are *not* explicitly generated.[1] Only after the above test passes do we need to compare the tuples in the projected tuple-sets.

Algorithm 3 Frequent graphs—Phase 3
Notation: L_n: set of composition tuple-sets of width n.
C_n is a candidate set for these compositions.

1. Set $n = 3$, $C_n = \emptyset$, $L_n = \emptyset$.

1. The fact that nonisomorphic paths can have the same descriptor is a complication, but not a serious problem, especially in labeled and directed graphs, where such cases are less likely to occur.

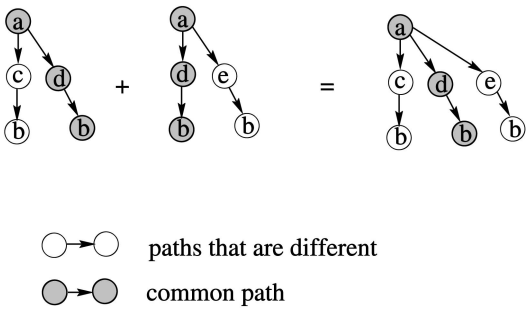

Fig. 8. Phase 3 example.

2. For every pair τ_1, τ_2 of (arbitrarily permuted) composition tuple-sets from L_{n-1} s.t. $\tau_1[1:n-2] = \tau_2[1:n-2]$, do:
   (a) Construct $\tau = BS(\tau_1, \tau_2)$. If $\Omega(\tau)$ is connected and has path number n, add τ to C_n.
   (b) For every composition tuple-set $S \in L_2$, if $\Omega(Splice(\tau, S))$ is connected and has path number n, add $Splice(\tau, S)$ to C_n.
3. Remove from C_n all composition tuple-sets that are not P-minimal.
4. For every $\tau \in C_n$, add τ to L_n if $\Omega(\tau)$ is frequent.
5. If $L_n = \emptyset$, halt.
6. Output $\{\Omega(\tau) | \tau \in L_n\}$, then set $n := n + 1$ and go to step 3.

In case 2b, *any* combination S of the two paths P_1, P_2 that is frequent and isomorphic to the remaining paths is found from L_2. Although not stated in the pseudocode, this step is fast because L_2 can be indexed for fast retrieval of compositions containing specific paths. The paths P_1, P_2 in the graph are combined (using *Splice*) with the generated candidate. This latter step is needed because merging two patterns directly (using bijective sum) may overlook cases where some nodes in the remaining paths are shared. Step 3 removes redundant isomorphic graphs, while step 4 checks the support of the candidates, as in phases 1 and 2.

An optional final step in the algorithm (not shown here) is removing all frequent subgraphs which are not maximal, i.e., contained in larger frequent graphs. Fig. 8 demonstrates merging two 2-path graphs that have one path in common into one 3-path graph.

3.2 Proof of Correctness

It is obvious by construction that our algorithm is sound since (in all phases) only frequent patterns are kept at the end of the computation. Therefore, showing completeness of the algorithm, i.e., that all frequent patterns are indeed found by the algorithm, is sufficient to prove correctness. Since all phases of the algorithm are separate (and run sequentially), completeness of each will be formally stated and proved separately.

Theorem 3. *When phase 1 (Algorithm 1) completes, L_1 contains all frequent single-path graph patterns.*

Proof outline. Note that from every path with k edges, an edge can be removed so that the remaining graph is a path with $k-1$ edges. Using the admissibility of the support measure, and the assumption that all frequent paths with $k-1$ edges were found in the previous iteration, the theorem follows by induction. □

Theorem 4. *Phase 2 (Algorithm 2) outputs all connected frequent graphs patterns with path number 2. Additionally, at the end of phase 2, the set L_2 contains all P-minimal composition tuple-sets, for every connected frequent graph pattern with path number 2.*

Proof. Let G be a frequent graph pattern with $p(G) = 2$. Then, G can be decomposed into two edge-disjoint paths and has a P-minimal decomposition P_1, P_2. Since we are using an admissible support measure, P_1 and P_2 are frequent and, by Theorem 3, an isomorphic copy of each of them is in L_1 at the end of phase 1. Denote the isomorphisms of P_1, P_2 by P_1', P_2', respectively. During phase 2, all possible consistent composition tuple-sets τ with $G(\tau) = (P_1', P_2')$ are constructed, including the composition τ for which $\Omega(\tau)$ is isomorphic to G under the natural isomorphism. Since the path descriptors are invariant under isomorphism and the decomposition of G into P_1, P_2 is P-minimal, then τ is also P-minimal and thus not pruned from C_2 at step 3. Since G is frequent, τ is stored in L_2 in step 4, and G is output at step 5. □

Theorem 5. *Phase 3 outputs all frequent connected graph patterns G with path number $p(G) \geq 3$.*

Proof outline. We show the invariant that, at the end of each iteration n, if G is a frequent graph with path number n, then there is a P-minimal composition $\tau \in L_n$ such that $\Omega(\tau)$ is isomorphic to G. Proof of the invariant is based on the invariant holding for graphs with path number $n-1$ at the beginning of the iteration, which holds for $n = 2$ due to Theorem 4. Using the admissibility of the support measure, we show that if G is frequent, then there exist P-minimal compositions in L_{n-1} with a common core of width $n-2$. These compositions induce frequent subgraphs G_1, G_2 with path number $n-1$ that are composed in the iteration by using bijective sum and splice to form G. □

3.3 Complexity Discussion

The complexity of our algorithm is composed of two components. The first component has to do with the problem definition and not with the specific algorithm. This complexity is exponential in the size of the pattern, and inherent to Apriori-like algorithms. The complexity of Apriori is due to the fact that the *number* of frequent patterns can be exponential and the complexity of any graph mining algorithm is constrained by the need to find all subgraphs of a database isomorphic to a given pattern in order to evaluate its support. The main goal of a mining algorithm should thus be to decrease the number of candidate patterns and, by doing so, decrease the number of support computations. Our approach is feasible because the number of patterns remaining from one phase to the next is reduced considerably, according to our experiments. The generation of candidate set C_{n-1} in the worst case, takes time:

$$O\left(\frac{|L_n|^2}{2} * n^2 * |L_2|\right).$$

In real-life cases, frequent patterns from the set L_n usually have different path structure and labeling and the number of candidate patterns created is much smaller. Even though the complexity is bounded by an exponential in n, in

reality, for large databases, the scan of the database whose complexity is $n * N$ may be worse and, in these cases, the approach of Apriori-TID may be beneficial.

3.3.1 Support Computation

The second component of complexity is due to the need to find all subgraphs isomorphic to the given pattern, which is exponential in the size of the pattern as well. While the large number of support computations is inherent to the basic Apriori algorithm ([1]), complexity of a single support computation is significantly higher for semistructured databases. Such computation requires 1) finding all subgraphs of a database isomorphic to a given graph pattern and 2) evaluating support using an admissible support measure. Finding all subgraphs of a database graph isomorphic to a given pattern depends strongly on the topology of a database graph. For a dense graph, the number of such subgraphs can be exponential to the size of a pattern. For a complete graph and an appropriate support value, every subgraph of a complete graph can be frequent! However, for a sparse graph (or the case for the real-life semistructured databases), the number of instances of a pattern is much smaller. In addition, a database graph with a large number of different labels is likely to produce a smaller number of pattern instances than a similar graph with a small number of different labels.

A formal complexity analysis of the entire algorithm is very difficult and thus not pursued here. Although the complexity is exponential in the worst case, the experiments in the next section suggest that, for nondense graphs, the algorithm is still reasonable for large graphs.

4 Empirical Evaluation

In the empirical evaluation, two sets of experiments were performed. The first set of experiments compares our algorithm to an edge-based algorithm. Two types of databases were used: synthetic, where we can control both the topology and labeling of graphs, and a real-life XML "movies" database [28]. Only the single graph setting was tested in this set of experiments. The second set of experiments compared our algorithm to FSG for both transactions and single graph settings. This set of experiments used also two databases, one synthetic, and one a real-life social network composed of electronic mails. The database records e-mails over a period of a week among users of the Ben-Gurion University e-mail system. The source, destination, and size of the message were recorded. The message size is used as an approximate "label" on the edge.

4.1 Experimental Setting

The experimental environment is a Sun Ultra-30 workstation running at 247 MHz and with 128 MB of main memory. The real XML file we used is a portion of the "movies" database. XML elements are treated as nodes and inheritance relationships and references as edges.

4.1.1 The Support Measure

The standard measure of support for transaction databases in the literature is as follows: The support S for an item set $I = \langle i_1, \ldots, i_k \rangle$ in a data set of transactions D is

$$S(I) = \frac{|\{t | t \in D, < i_1, \ldots, i_k > \in t\}|}{|D|}. \quad (2)$$

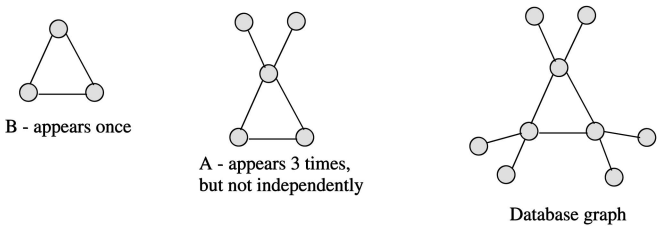

Fig. 9. Graph pattern support.

However, if the application makes it necessary to count the total number of occurrences of a pattern, the above scheme is inappropriate. An alternate definition of support, taking the multiple occurrences into account, must be defined, a nontrivial issue due to possible overlaps between instances.

For example, one trivial support measure is the number of instances of a frequent pattern. This measure, however, is not admissible. Fig. 9 shows a database that contains three instances of pattern A and only one instance of pattern B, while $B \subseteq A$. Another approach is to take into account all automorphisms of a pattern in question. Again, the database in Fig. 9 is a counterexample, since $|Aut(B)| = 6$ and $|Aut(A) = 4|$, making the total count of A's instances 12, which is still greater than six.

The only nontrivial provably admissible measure we could find for the single graph setting is defined as follows [32]: Let D be a database graph and G be a graph pattern for which we wish to compute support. Let $A_1, A_2, \ldots A_n$ be all instances of G in D. We create a new graph called the *instance graph*, in which each of the A_i is a node and there is an edge between A_i and A_j if the two subgraphs A_i and A_j have at least one common edge. The maximum independent set (MIS) measure is defined as the size of the maximum independent set over the instance graph and was shown in [32] to be admissible.

Using the MIS measure, we must compute the maximum independent set of the instance graph I_G. Theoretically, this can take time exponential to the size of I_G, since the independent set problem is NP-hard. However, for real-life cases of sparse database graphs with a reasonable number of labels, this task is usually much easier. In our experiments, time for computing the maximum independent set was actually negligible compared to the time to find the instances. Therefore, the performance of the algorithms is not strongly dependant on the specific (MIS) support measure. In addition, approximation techniques can be used in this case (see [14] for details) as a user usually does not care about a *precise* support value.

4.2 The Implemented Algorithms

We implemented the mining algorithm for fully labeled graphs described in Section 3, as well as the two types of edge-based algorithms discussed below. The latter were used in order to compare the number of generated candidate patterns with our algorithm.[2] The same admissible MIS support measure was used for all algorithms.

2. The reason our comparison is done opposite simple edge-based algorithms, rather than to FSG or GSPAN, is that the latter algorithms use the transaction-graph setting, making a direct comparison inapplicable. Additionally, little research exists on algorithms that use the maximum independent set (MIS) support measure, the only non-trivial admissible support measure we know for the single-graph setting (see Section 5).

TABLE 4
Notation Used in Results Tables

N, E, L	# of nodes, edges and labels in the database
S	support threshold in %
C, FP	# of candidate and frequent patterns
I, SC	# of isomorphism and support computations
TT	total time (seconds) spent on data mining
ST	time in sec. of support computations
EA	edge addition algorithm
PM	Path Mining, denotes our algorithm
#	serial number of a database graph
CR	candidate ratio

Tests were conducted multiple times and time averages were taken to eliminate factors of system load.

The first algorithm is based on finding all frequent graphs G with k edges and then extending each graph G into graphs with $k+1$ edges by either adding a new node and an edge to G frequent graph or by adding an edge between two existing nodes of G. The process is repeated until no graphs extended in this manner are frequent. For the comparison with FSG, we have implemented a version of FSG for the single graph setting, based on [20].

4.3 Experimental Results

4.3.1 First Set

We investigated the behavior of the algorithms using the following performance parameters: 1) *number of candidate patterns* produced by an algorithm during data mining, 2) *number of isomorphism computations* during data mining and overall number of support computations, and 3) *total time* spent on data mining (**not** CPU time) and on support computations. Table 5 presents results for testing on synthetic trees and synthetic sparse graphs. The notation used in all three tables is explained in Table 4. For our algorithm, the number of candidate patterns can sometimes be less than the number of frequent patterns since frequent nodes and edges are computed directly without generating candidate patterns. Our implementation needs to generate all appropriate subgraphs of a database graph, find among them all subgraphs that are isomorphic to the pattern in question, and build an instance graph and find its maximum independent set size. Thus, testing our algorithm on dense graphs seems to be extremely time consuming. An additional consideration was the fact that most real-life databases represent sparse graphs rather than dense ones. Therefore, we decided to limit our tests to trees and sparse graphs and to choose a support threshold that, on the one hand, will not limit the output to trivial graphs (nodes and edges) and on the other hand, will not make every connected subgraph of the database frequent.

From Table 5, we conclude that our algorithm runs faster even though it conducts more isomorphism checks than the edge addition algorithm. The latter occurs because our algorithm produces fewer candidate patterns and, thus, less time is wasted on support computation.

Table 6 contains the number of frequent patterns found in six different subsets of the movie database with different support values. The structure of the database (a tree as in set number 6 or a sparse graph) can be seen to have more impact on the number of frequent patterns than the support value.

As seen from Table 6, for the same values of support, the number of frequent patterns is smaller and thus the execution time is *much smaller* in the movie database than in the synthetic data set. This indicates the feasibility of our algorithm in real-life cases. As the graph becomes larger, the number of frequent patterns for the same support value decreases since a larger number of edge-disjoint instances is required for each pattern in order to pass the support threshold. Note that these patterns do not contain titles of movies or names of directors, since these are present only as *attributes* and not as *tags* in the XML database. Related research [25] attempts to treat attributes and values of an XML database as well.

TABLE 5
Experimental Results for Trees and Sparse Graphs

Trees						Sparse graphs				
#	N, L, S, FP	Alg	C, I, SC	ST	TT	N, E, L, S, FP	Alg	C, I, SC	ST	TT
1	40 4 7% 15	EA	100 24 92	41	44	40 50 4 7% 14	EA	60 33 52	19	24
		PM	52 47 52	12	20		PM	49 55 42	14	23
2	50 4 7% 16	EA	110 41 102	676	682	40 50 6 5% 17	EA	84 48 76	33	40
		PM	45 45 42	22	29		PM	59 70 54	15	26
3	50 6 3% 37	EA	470 82 458	326	340	50 60 6 5% 28	EA	355 74 343	314	326
		PM	202 239 205	68	106		PM	117 185 143	41	70
4	50 8 3% 27	EA	306 62 290	280	290	60 80 4 4% 16	EA	101 31 93	609	614
		PM	119 91 111	12	39		PM	56 58 56	123	132
5	60 4 5% 15	EA	100 24 92	220	224	60 80 6 3% 27	EA	265 86 253	842	857
		PM	52 47 52	56	63		PM	120 102 110	41	57
6	60 6 5% 44	EA	728 203 716	3493	3537	70 90 8 3% 27	EA	252 77 236	44	57
		PM	175 868 276	238	376		PM	126 98 110	21	37
7	60 8 5% 14	EA	103 18 87	19	22	80 100 8 3% 32	EA	403 74 387	160	172
		PM	41 29 26	4	9		PM	149 127 141	41	60

TABLE 6
Movie Database: Support versus Frequent Patterns

Data set	Nodes	Edges	Labels	S	50%	40%	30%	20%	10%	9%	8%	7%	6%	5%
#1	12656	13878	112	FP	5	6	7	8	15	16	16	18	21	22
#2	8337	9416	25	FP	3	4	5	6	12	12	12	12	13	14
#3	7027	7851	22	FP	3	4	4	5	10	10	10	10	11	11
#4	4730	4813	90	FP	5	5	8	9	11	11	12	12	12	16
#5	2757	2794	76	FP	2	2	5	6	7	7	8	9	10	11
#6	1293	1292	91	FP	21	32	34	46	79	79	84	84	86	86

We deduce the following facts from our experiments:

1. Our algorithm produces fewer candidate patterns and therefore performs fewer support computations than the edge addition algorithm.
2. Support computation is easier if the database is a tree due to fewer candidate patterns.
3. Synthetic graphs are not very regular. As the number of distinct labels in synthetic database increases, the chance of finding nontrivial frequent patterns in that database decreases drastically.
4. Large real-life graph databases are highly regular and contain complex patterns.

4.3.2 Second Set—Comparison with FSG

In this set of experiments, we compared FSG with our algorithm for both transaction setting and single graph setting. For the transaction setting, the results were comparable and are not shown here. For the single graph setting, we measured both the time and the number of support computations. Since the running time was dominated by the number of support computations, we decided not to report it at all and, instead, report the number of support computations, which is equal to the number of candidates generated. Therefore, in all the tables and graphs below, the measure of efficiency is the number of candidates generated.

Table 7 shows numbers of candidates and frequent patterns generated by both algorithms for various support values on two subsets (5,000 and 2,000 nodes) of a Ben-Gurion University e-mail traffic database. The entire database is large (over 50,000 nodes) and quite dense, which makes it difficult to mine. In all tables, PM stands for *Path Mining* and denotes results achieved by our algorithm.

TABLE 7
BGU E-Mail Database Results

5000 nodes					2000 nodes				
S	C FSG	C PM	FP	CR	S	C FSG	C PM	FP	CR
1%	54	45	9	1.2	0.1%	209	190	19	1.1
0.9%	54	45	9	1.2	0.09%	252	231	21	1.09
0.8%	65	55	10	1.18	0.08%	252	231	21	1.09
0.7%	65	55	10	1.18	0.07%	275	253	22	1.09
0.6%	65	55	10	1.18	0.06%	275	253	22	1.09
0.5%	65	55	10	1.18	0.05%	405	378	27	1.07
0.4%	77	66	11	1.17	0.04%	405	378	27	1.07
0.3%	77	66	11	1.17	0.03%	527	496	31	1.06
0.2%	119	105	14	1.13	0.02%	527	496	31	1.06
0.1%	230	210	20	1.1	0.01%	1034	990	44	1.04

Table 8 shows numbers of candidates and frequent patterns generated by both algorithms for various support values on random graphs with 3,000 nodes, 4,000 edges and different numbers of labels: 30, 40, and 50. These results show that our algorithm produces fewer candidate patterns than FSG and therefore performs fewer support computations.

Fig. 10 shows numbers of candidates generated by both algorithms for various support values on random graphs with 1,000 nodes, 2,000 edges and different numbers of labels: 10 and 20, respectively. We learned from our experiments that support computation is the factor having the most impact on the computation time because of the need for multiple subgraph isomorphism computations in both single and multiple graph settings. Reducing support computation is significantly more important than computing a DFS code of a pattern or eliminating isomorphic candidates, since frequent patterns are not very large compared to the database size.

5 Discussion and Related Work

This section briefly presents related work and discusses our contribution in the context of prior research in the field. As mentioned in the introduction, most of the work done on graph mining is comparatively recent. The basic work related to this subject is frequent itemset mining in structured databases and the Apriori algorithm and its variations [1]. For conciseness, reference to the significant body of existing work on transaction database mining is omitted. Papers that deal with mining topologically simple patterns, such as paths and trees, are directly related to our work and thus reviewed below.

Paper [2] presents two algorithms for mining frequent directed simple path patterns in a Web environment. Both algorithms are based on an algorithm called MF that finds all maximal forward references in a set of traversal sequences contained in the database. The goal of the two mining algorithms is to find frequent sequences in these paths. The main differences between the algorithm of [2] and ours is that the former handles only linear paths, making its support measure computationally simple.

The simple paths mining problem is generalized in [36], which describes an algorithm for finding maximal frequent treelike patterns in semistructured documents, represented in the standard OEM model. Although this algorithm searches only for treelike patterns, it can also handle patterns containing cycles by transforming them into trees.

TABLE 8
Random Graph with 3,000 Nodes and 4,000 Edges

50 labels				40 labels				30 labels			
S	C FSG	C PM	FP	S	C FSG	C PM	FP	S	C FSG	C PM	FP
0.2	113	83	28	0.3	28	20	8	0.5	3	2	1
0.19	113	83	28	0.25	187	152	35	0.45	9	6	3
0.18	113	83	28	0.24	187	152	35	0.4	40	29	11
0.17	306	244	58	0.23	187	152	35	0.35	185	147	32
0.16	306	244	58	0.22	408	339	63	0.3	884	707	92
				0.21	408	339	63				
				0.2	1216	1020	126				

One important restriction in this paper is that only *rooted* trees are considered, i.e., trees whose root is the same as the root of the entire Web database. Chi et al. handle the problem of tree mining in a wider sense in [27].

Work on mining general graph patterns began in the 1990s. A recent survey of graph mining, by Washio and Motoda [37], presents some of the earlier works on the subject like SUBDUE [7] and GBI [41]. It then classifies the mining algorithms into two major categories: Greedy search algorithms, which search exhaustively for all the frequent graph patterns, and Inductive (ILP) approaches, which pregenerate many graph patterns according to some logic constraints and background knowledge and then use a query language to retrieve the interesting patterns [26]. Since our paper uses the greedy approach, we do not further discuss ILP here.

Regarding the greedy approach, two categories of algorithms were mentioned in the introduction: transaction graphs and single graph settings. To date, most work has been on the transaction graph setting, with algorithms divided roughly into two classes: breadth-first search (or Apriori-based) and depth-first search.

Most BFS algorithms use the basic idea employed in the Apriori algorithm. The main difference between the various algorithms of this category is in the type of the *building block* used to generate the item of level K. Inokuchi et al. [17] use vertices. An algorithm by Kuramochi and Karpis [19] uses edges as the main building block and was extended and improved in [20] by adding several clever heuristics that make mining and support computation more efficient. This latter version, called FSG, is currently one of the best known and often compared to a version of the BFS graph mining algorithms for the graph-transaction setting case. FSG introduces the definition of a canonical labeling of graphs based on the adjacency matrix, used to eliminate isomorphic candidates. To increase the efficiency of deriving the canonical labels, the approach uses some graph vertex invariants, such as the degree of each vertex in the graph. FSG also increases the efficiency of the candidate frequent subgraph generation by introducing the transaction ID (TID) method. Furthermore, FSG limits the class of the frequent subgraphs to connected graphs. Under this limitation, FSG introduces an efficient search algorithm using a "core," which is a shared part of size $k-1$ in the two frequent subgraphs of the size k. FSG increases the joining efficiency by limiting the common part of the two frequent graphs to the core. Once the candidates are obtained, their frequency counting is conducted by checking the cardinality of the intersection of both TID lists. FSG is fast due to the introduction of numerous techniques, but its memory consumption is heavy (storage for TID lists of massive graph data). Some ideas similar to those in FSG, e.g., those related to joining of two subgraphs, are present in

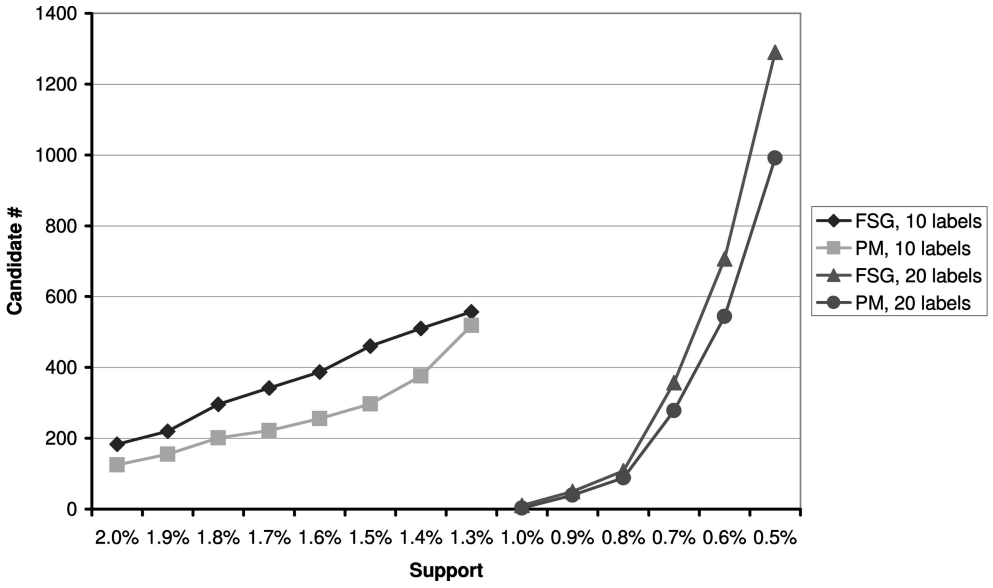

Fig. 10. Random graph with 1,000 nodes and 2,000 edges.

this paper as well. However, the method in this paper was derived independently and our use of edge-disjoint paths as a building block is new. Other works which use the BFS approach are [18], [15], [16], [6].

The second approach, called gSpan [39], grows patterns from a single graph directly, by using a depth-first search strategy. The algorithm maps each pattern to a unique canonical label and assigns each graph a unique minimum DFS code. By using these labels, a complete order relation is imposed over all possible patterns. This lexicographic order is also used to impose a tree-hierarchy order over all patterns, resulting in a hierarchical search tree. This search-tree is traversed in a DFS manner, pruning on the way all subgraphs with nonminimal DFS code. This algorithm also uses the TID approach. Since the algorithm explores the search space in DFS manner, it enables the use of several mining techniques which are especially applicable to DFS algorithms, such as maintaining an embedding set for each frequent subgraph, like [53]. Yan and Han [39] also present an experimental evaluation, where they compare gSpan with FSG and show the better performance of gSpan on several molecular databases. Several of the ideas of [39] were used later, in an approach which is intermediate between BFS and DFS, in [17].

In summary, the ideas presented in the above papers have influenced our work considerably. However, using paths as building blocks and an efficient method for merging graphs represented as compositions of paths are original contributions of this paper. From our experiments, we did not see an inherent problem of scaling up the algorithm to very large graphs, other than memory requirements encountered with large graphs. These may be handled similarly to [34].

6 Conclusion

An Apriori-like algorithm for retrieving frequent graph patterns from a given set of graphs is the central issue in this paper. In contrast with most existing work, the pattern can be either a directed or an undirected graph and may contain cycles. The added functionality can support data mining on the increasing fraction of online documents that consist of blocks connected by references. Knowledge about typical structure of documents is helpful in analyzing complex repositories of semistructured data (e.g., XML databases, the Web) and is potentially useful for querying data, indexing it, and storing it efficiently. In searching for frequent patterns, candidates are constructed using frequent paths. The scheme is evaluated empirically and is promising as it shows a decided advantage over other algorithms. The scheme proposed here can be extended in several ways, such as using partially labeled patterns, using more complex building blocks (trees), adapting the algorithm to the dynamic database model, and using the Apriori-TID technique.

Acknowledgments

This research was partially supported by the KITE consortium under contract to the Israeli Ministry of Trade and Industry and by the Paul Ivanier Center for Robotics and Production Management. The authors wish to thank Marina Litvak for implementing a significant fraction of the code for the experiments, and the anonymous reviewers for useful comments that contributed to the final version of the manuscript.

References

[1] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules," *Proc. 20th Int'l Conf. Very Large Data Bases,* Sept. 1994.
[2] M.S. Chen, J.S. Park, and P.S. Yu, "Efficient Data Mining for Path Traversal Patterns," *IEEE Trans. Knowledge and Data Eng.,* vol. 10, no. 2, pp. 209-221, Mar./Apr. 1998.
[3] D. Chamberlin, "XQuery: A Query Language for XML," *Proc. SIGMOD Conf.,* 2003.
[4] Y. Chi, S. Nijssen, R.R. Muntz, and J.N. Kok, "Frequent Subtree Mining: An Overview," *Fundamenta Informaticae,* special issue graph and tree mining, 2005.
[5] C. Chung, J. Ki Min, and K. Shim, "APEX: An Adaptive Path Index for XML Data," *Proc. SIGMOD Conf. 2002,* pp. 121-132, 2002.
[6] M. Cohen and E. Gudes, "Diagonally Subgraphs Pattern Mining," *Proc. Ninth ACM SIGMOD Workshop Research Issues in Data Mining and Knowledge Discovery,* 2004.
[7] J. Cook and L. Holder, "Substructure Discovery Using Minimum Description Length and Background Knowledge," *J. Artificial Intelligence Research,* pp. 231-255, 1994.
[8] L. Dehaspe, H. Toivonen, and R.D. King, "Finding Frequent Substructures in Chemical Compounds," *Proc. Fourth Int'l Conf. Knowledge Discovery and Data Mining (KDD '98),* pp. 30-36, 1998.
[9] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, D. Maier, and D. Suciu, "Querying XML Data," *IEEE Data Eng. Bull.,* vol. 22, no. 3, pp. 27-34, 1999.
[10] A. Deutsch, M.F. Fernandez, and D. Suciu, "Storing Semistructured Data with STORED," *Proc. SIGMOD Conf.,* pp. 431-442, 1999.
[11] C. Domshlak, R. Brafman, and S.E. Shimony, "Preference-Based Configuration of Web Page Content," *Proc. Int'l Joint Conf. Artificial Intelligence,* Aug. 2001.
[12] L. Garton, C. Haythornthwaite, and B. Wellman, "Studying Online Social Networks," *J. Computer-Mediated Comm.,* vol. 3, no. 1, 2004.
[13] R. Goldman and J. Widom, "DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases," *Proc. 23rd Very Large Data Bases Conf. (VLDB '97),* 1997.
[14] E. Gudes, S.E. Shimony, and N. Vanetik, "Support Measures for Semistructured Data," *Data Mining and Knowledge Discovery J.,* to appear in vol. 13, 2006.
[15] M. Hong, H. Zhou, W. Wang, and B. Shi, "An Efficient Algorithm of Frequent Connected Subgraph Extraction," *Proc. Pacific-Asia Conf. Knowledge Discovery and Data Mining (PAKDD),* 2003.
[16] J. Huan, W. Wang, and J. Prins, "Efficient Mining of Frequent Subgraphs in the Presence of Isomorphism," *Proc. IEEE Int'l Conf. Data Mining (ICDM '03),* pp. 549-552, 2003.
[17] A. Inokuchi, T. Washio, and H. Motoda, "An Apriori Based Algorithm for Mining Frequent Substructures from Graph Data," *Proc. European Conf. Principles of Data Mining and Knowledge Discovery (PKDD '00),* 2000.
[18] A. Inokuchi, T. Washio, and H. Motoda, "Complete Mining of Frequent Patterns from Graphs, Mining Graph Data," *Machine Learning,* vol. 50, no. 3, pp. 321-354, 2003.
[19] M. Kuramochi and G. Karypis, "Frequent Subgraph Discovery," *Proc. IEEE Int'l Conf. Data Mining (ICDM),* 2001.
[20] M. Kuramochi and G. Karypis, "An Efficient Algorithm for Discovering Frequent Subgraphs," *IEEE Trans. Knowledge and Data Eng.,* vol. 16, no. 9, Sept. 2004.
[21] M. Kuramochi and G. Karypis, "Finding Frequent Patterns in a Large Sparse Graph," *Proc. 2004 Soc. Industrial and Applied Math. (SIAM) Data Mining Conf.,* 2004.
[22] V. Lipets and E. Gudes, "An Efficient Algorithm for Subgraph Isomorphism," *Proc. Fourth Haifa Workshop Graph Theory and Algorithms,* 2004.
[23] X. Lin, C. Liu, Y. Zhang, and X. Zhou, "Efficiently Computing Frequent Tree-Like Topology Patterns in a Web Environment," *Proc. 31st Int'l Conf. Technology of Object-Oriented Language and Systems,* 1998.
[24] A. Meisels, M. Orlov, and T. Maor, "Discovering Associations in XML Data," technical report, Ben-Gurion Univ., 2001.
[25] T. Milo and D. Suciu, "Index Structures for Path Expressions," *Proc. Int'l Conf. Database Theory (ICDT '99),* pp. 277-295, 1999.
[26] S. Muggleton and L. DeRaedt, "Inductive Logic Programming: Theory and Methods," *J. Logic Programming,* vol. 19, no. 2, pp. 629-679, 1994.
[27] S. Nijssen and J.N. Kok, "Frequent Graph Mining and Its Application to Molecular Databases," *Proc. IEEE Int'l Conf. Systems, Man, and Cybernetics,* pp. 4571-4577, 2004.
[28] *Internet Movie Database,* http://us.imdb.com, 2002.

[29] X. Pennec and N. Ayache, "A Geometric Algorithm to Find Small but Highly Similar 3D Substructures in Proteins," *Bioinformatics,* vol. 14, no. 6, pp. 516-522, 1998.
[30] D. Shasha, J.T.L. Wang, and R. Guigno, "Algorithmics and Applications of Tree and Graph Searching," *Proc. 21st ACM SIGMOD-SIGACT-SIGART Symp. Principles of Database Systems,* pp. 39-52, 2002.
[31] F. Tian, D. DeWitt, J. Chen, and C. Zhang, "The Design and Performance Evaluation of Alternative XML Storage Strategies," technical report, Computer Sciences Dept., Univ. of Wisconsin, 2000.
[32] N. Vanetik, E. Gudes, and S.E. Shimony, "Computing Frequent Graph Patterns from Semistructured Data," *Proc. Int'l Conf. Data Mining (ICDM),* pp. 458-465, 2002.
[33] N. Vanetik and E. Gudes, "Mining Frequent Labeled and Partially Labeled Graph Patterns," *Proc. Int'l Conf. Data Eng. (ICDE '04),* pp. 91-102, 2004.
[34] C. Wang, W. Wang, J. Pei, Y. Zhu, and B. Shi, "Scalable Mining of Large Disk Based Graph Database," *Proc. Int'l Conf. Knowledge Discovery and Data Mining (KDD '04),* 2004.
[35] K. Wang and H. Liu, "Discovering Typical Structures of Documents: A Road Map Approach," *Proc. SIGIR Conf.,* pp. 146-154, 1998.
[36] X. Wang, J.T. Li Wang, D. Shasha, B. Shapiro, I. Rigoutsos, and K. Zhang, "Finding Patterns in Three-Dimensional Graphs: Algorithms and Applications to Scientific Data Mining," *IEEE Trans. Knowledge and Data Eng.,* vol. 14, no. 4, pp. 731-749, July/Aug. 2002.
[37] T. Washio and H. Motoda, "State of the Art of Graph-Based Data Mining," *SIGKDD Explorations,* July 2003.
[38] S. Wasserman, K. Faust, and D. Iacobucci, *Social Network Analysis: Methods and Applications (Structural Analysis in the Social Sciences).* Cambridge Univ. Press, 1994.
[39] X. Yan and J. Han, "gSpan: Graph-Based Substructure Pattern Mining," *Proc. Int'l Conf. Data Mining,* pp. 721-724, 2002.
[40] X. Yan and J. Han, "CloseGraph: Mining Closed Frequent Graph Patterns," *Proc. ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining (KDD '03),* 2003.
[41] K. Yoshida, H. Motoda, and N. Indurkhya, "Graph-Based Induction as a Unified Learning Framework," *J. Applied Intelligence,* pp. 297-328, 1994.

Ehud Gudes recieved the BSc and MSc degrees from the Technion and the PhD degree in computer and information science from the Ohio State University in 1976. Following his PhD, he worked both in academia (Pennsylvania State University, Ben-Gurion University (BGU),), where he did research in the areas of database systems and data security, and in industry (Wang Laboratories, National Semiconductors, Elron, and IBM Research), where he developed query languages, CAD software, and expert systems for planning and scheduling. He is currently an associate professor in computer science at BGU, and his research interests are knowledge and databases, data security and data mining, especially graph mining. He is a member of the IEEE Computer Society.

Solomon Eyal Shimony received the BSc degree in electrical engineering from the Technion in 1982 and the PhD degree in computer science from Brown University in 1991, after which he joined the Department of Computer Science at Ben-Gurion University (BGU). At present, he is a deputy head of the Computer Sciences Department at BGU, chair of the Paul Ivanier Center for Robotics and Production Management, and an associate editor of the *IEEE Transactions on Systems, Man, and Cybernetics, Part B*. His research interests are artificial intelligence, probabilistic reasoning, knowledge discovery in databases, robotics, flexible computation, and spatial data models. He is a member of the IEEE Computer Society.

Natalia Vanetik received the BSc degree in mathematics and computer science from Ben-Gurion University in 1996 and the MSc degree in mathematics and computer science from Ben-Gurion University in 2003. She is currently a PhD student with the Department of Computer Science at Ben-Gurion University. Her research interests include combinatorial optimization, graph theory, and graph mining.