Discovering Frequent Graph Patterns Using Disjoint Paths

E. Gudes, S. E. Shimony, N. Vanetik
{ehud,shimony,orlovn}@cs.bgu.ac.il
Dept. of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, Israel

Abstract
Whereas data-mining in structured data focuses on frequent data values, in semi-structured 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 sub-tasks, especially sub-graph 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.

Keywords: H.2.8 Database Applications: H.2.8.d Data mining, H.2.8.i Mining methods and algorithms, H.2.8.m 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 semi-structured data such as web data ([24],[36],[2]), chemical compounds data [8, 28] or biological data [30]. 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 movie XML database [29], some movies have more actors than others; some fields (e.g., Award) are missing for some movies; some actors have birthday 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% of Actor objects have a Nationality label, etc. This phenomenon is common in other types of semi-structured data as well [20].

While in the field of structured data-mining frequent data values and their common appearances are of interest, in mining semi-structured data, the focus is on frequent labels and common appearances of sub-sets 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, or on edges, or on both (the transformation between these types of models 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 Figure 1, depicting some frequent graph patterns found by our algorithm in an XML movie database [29].

Figure 1: Pattern examples

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, 32]. 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...
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 and XML data [13, 26, 14]. 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, 39].

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 problems tree and path mining are most relevant [4]. However, if one looks at sets of related web navigation patterns, or at behaviour over time, one gets more complex patterns which can be represented by graphs, motivating the use of graph mining. Another application related to user behaviour is the area of social networks, analysing which is an important field in communication and in security applications [12]. An example social network database based on e-mails is used in our study.

In the past, most work done in this field dealt with either single path patterns [2] or tree-like patterns [24, 36, 4]. However, much of the data on the web is graph-like, 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 [18], FSG [21], gSpan [40], CloseGraph [41], and AdiMine [35]. In this paper we present a new algorithm for mining frequent patterns in semi-structured 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 datasets. In the former, 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 sub-graphs in molecular transactions [21].

In the alternate 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 [22], 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 [20, 21, 40, 19, 16, 17, 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 datasets. 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 datasets and application domains than the former problem. Other than our own papers [33, 34], the most recent paper dealing with the single graph setting is [22], discussed in Section 5.
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 dataset 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 dataset graphs to real numbers (usually in $[0, 1]$). As usually the dataset graph is understood, this argument to $S$ is omitted. $S$ obeys the admissibility constraint (also called anti-monotonicity, or downward closure) if every subgraph of a frequent pattern is also frequent [1, 15]. 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 dataset. Eliminate the non-frequent patterns.

2. Find all candidate patterns composed of two frequent paths, and eliminate the non-frequent patterns.

3. At each successive step $n$:
   - (a) Construct candidate patterns from smaller frequent ones, which 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 Figure 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.

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 is due to involves multiple computations of the sub-graph 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
order to create larger graphs, but are not concerned about how to traverse the paths once they have been created. Henceforth, we ignore the order 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 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:

2. For every connected undirected graph $G = (V, E)$, $p(G) = 1$ if $G$ is Eulerian, and $p(G) = |\{v \mid v \in V, d(v) \text{ is odd}\}|/2$ 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 $\text{label}(v)$. We assume without loss of generality that the dataset (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 $\phi : V' \rightarrow V''$ such that for every $v \in V'$, $\text{label}(v) = \text{label}(\phi(v))$. When such an isomorphism exists, denote by $G' \cong G''$ 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 dataset labeled graph $G$, and a support measure $\sigma$ over pattern graphs, and a support threshold $\sigma$, 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 w.l.o.g. through.

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
\[ RT_P(v) := (\text{label}(v), d^+_P(v), d^-_P(v)) \]

For undirected paths, the representing tuple is likewise defined as:
\[ RT_P(v) := (\text{label}(v), d_P(v)) \]

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 \( v \) and \( w \), denoted \( v <_L w \), is understood. Likewise, equality operator \( v =_L w \) denotes equality of the respective representing tuples.

Path \( P \) is covered by path \( Q \) if and only if \( Q \) is a path in the graph \( G \) that can be reached from the start node of \( P \) through \( P \) and \( Q \). The resulting sorted sequence, denoted by \( \text{lex}_L \), is the sorted sequence of the composition descriptors of all the minimal path covers of \( G \).

Theorem 2 Let \( G = (V, E) \) be a connected graph with \( p(G) \geq n \geq 2 \) and \( (P_1, \ldots, P_n) \) 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 \( \{v_i, v_j\} \in E' \) if and only if \( P_i \cap P_j \neq \emptyset \). 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...
Finally, a minimal path cover of a connected graph with path number greater than 1 consists only of non-cyclic 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 merged 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 non-cyclic 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 non-cyclic. Let \( P \in \mathcal{P} \), and 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. \( \square \)

### 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.

#### 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 \( i \)th 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 \( T[i:j] = \{ t[i:j] \mid t \in T \} \). A set subtraction operator inside the square brackets indicates removal of the subtracted elements from the tuple (resp. 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 (resp. 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 composi-

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 element-wise composition, thus: \( (a, \perp, b, \perp) + (a, b, \perp, \perp) = (a, b, b, \perp) \). A vector composition where any of the element-wise 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 \( G \) be a set of graphs. A composition tuple-set \( \tau \) of width \( n \) over \( G \) is a pair \((G(\tau), \text{tuples}(\tau))\), where \( G(\tau) \) is an \( n \)-tuple with each element designating a graph in \( G \), and \( T = \text{tuples}(\tau) \) 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 \( 1 \leq i < j \leq n \) 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 \( \text{tuples}(\tau) \) will be denoted by \( \text{width}(\tau) \). Observe that \( G(\tau) \) may have more than one element referring to the same graph. In our algorithm, the set \( G \) 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 \( G \) is the set of paths in our dataset graph, and thus omit reference to \( G \). (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 \( \tau \) (over \( G \)) is consistent if all the following conditions hold:

1. For every \( 1 \leq i \leq \text{width}(T) \) 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.)
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 $V = \{\nu(t) \mid t \in T\}$ (where $\nu$ is an arbitrary function that assigns a unique node to every tuple $t$), and $E = \{\{\nu(t_1), \nu(t_2)\} \mid t_1, t_2 \in T \land \exists i (t_1[i], t_2[i]) \in E(G(\tau)[i])\}$.

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 $\tau$ 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 $\nu$ evaluates to a new unique node, i.e. one that does not appear elsewhere in the system.

Figure 3 shows a graph consisting of 3 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.

**Figure 3:** Graph $G$ composed from 3 edge-disjoint paths: $P_1, P_2, P_3$.

<table>
<thead>
<tr>
<th>Node</th>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_1$</td>
<td>$a_1$</td>
<td>$\perp$</td>
<td>$\perp$</td>
</tr>
<tr>
<td>$v_2$</td>
<td>$a_2$</td>
<td>$b_2$</td>
<td>$\perp$</td>
</tr>
<tr>
<td>$v_3$</td>
<td>$a_3$</td>
<td>$\perp$</td>
<td>$\perp$</td>
</tr>
<tr>
<td>$v_4$</td>
<td>$\perp$</td>
<td>$b_1$</td>
<td>$\perp$</td>
</tr>
<tr>
<td>$v_5$</td>
<td>$\perp$</td>
<td>$b_3$</td>
<td>$c_3$</td>
</tr>
<tr>
<td>$v_6$</td>
<td>$\perp$</td>
<td>$\perp$</td>
<td>$c_1$</td>
</tr>
<tr>
<td>$v_7$</td>
<td>$\perp$</td>
<td>$\perp$</td>
<td>$c_2$</td>
</tr>
</tbody>
</table>

Table 1: Composition tuple-set $\tau$ on $P_1, P_2, P_3$.

**Observation 2:** Let graph $G$ be covered by $n$ mutually edge-disjoint subgraphs $G_1, G_2, ..., G_n$. Then $G$ is identical to the graph induced by the composition tuple-set $\tau = (G(T), T)$ constructed as follows: $G(T) = (G_1, G_2, ..., G_n)$, and tuples($T$) consists of $|V(G)|$ tuples, one unique tuple $t$ for each node in $G$, 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 w.r.t. $G$ and its cover.

**Proposition 1** The composition tuple-set $\tau$ 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 $\tau$ is P-minimal if there is no $\tau'$ such that $\Omega(\tau') = \Omega(\tau)$ and $G(\tau') <_{lex} G(\tau)$.

**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:

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

we indicate that $\tau'$ is a projection of the composition $\tau$ 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 $n$:

$$T' = T[1 : n \setminus i]$$

$$= (G(T)[1 : n \setminus i], \text{tuples}(T)[1 : n \setminus i] \setminus \perp)$$

The resulting $\tau'$ is a composition of width $n - 1$. 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, $\tau$ a consistent composition, and $t_1, t_2 \in \text{tuples}(\tau)$ with $t_1 \neq t_2$. Then $t_1[R] = t_2[R]$ implies $t_1[R] = t_2[R] = \perp$. 


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 $T_1 = \text{tuples}(\tau_1)$, and $T_2 = \text{tuples}(\tau_2)$. The bijective sum of $\tau_1$ and $\tau_2$, denoted $BS(\tau_1, \tau_2)$, is a composition $\tau$ 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. $\{t_1, t_2[n - 1] \mid t_1 \in T_1, t_2 \in T_2, t_1[1 : (n - 2)] = t_2[1 : (n - 2)] \neq \bot\}$

2. $\{\bot^{n-2}.t[n-1], \bot \mid t \in T_1, t[1 : (n - 2)] = \bot\}$ (where $\bot^i$ means an all-$\bot$ $i$-tuple).

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

The intuition for this definition is as follows, by considering the induced graphs of the composition tuple-sets (see Figure 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 $\tau_1$ and $\tau_2$. Tuples in (2) correspond to nodes that appear in the graph induced by $\tau_1$, but do not appear in $\tau_2$. Likewise, tuples in (3) correspond to nodes that appear in the graph induced by $\tau_2$, but do not appear in $\tau_1$. 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.

The definition of bijective sum can easily be generalized to allow for the equivalent part of $\tau_1$ and $\tau_2$ 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 $\tau_1$ and $\tau_2$ in order to get $\tau_1[1 : (n - 2)] = \tau_2[1 : (n - 2)]$, performing the bijective sum, and arbitrarily permuting the positions of $\tau$. 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 Figure 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 type (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 Figure 5 as an example).

Definition 5 (Splice) Let $\tau$ 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 $\tau$ by $S$, denoted $\text{Splice}(\tau, S)$, is a composition $\tau'$ with $G(\tau') = G(\tau)$, and $T' = \text{tuples}(\tau')$ defined as follows. Denote $T = \text{tuples}(\tau)$, $s = \text{tuples}(S)$, and let $M$ be a set of “merged” tuples:

$$M = \{ t_1 + t_2 \mid t_1, t_2 \in T, \exists s \in S$$

$$t_1[1 : n - 1] = s[1] \neq \bot \land t_2[n] = s[2] \neq \bot$$


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 Equation 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 $T_3 = \text{Splice}(T_1, T_2)$. Figure 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 2: Bijective sum.

<table>
<thead>
<tr>
<th>Node</th>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_3$</th>
<th>Node</th>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_3$</th>
<th>$P_4$</th>
<th>Node</th>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_3$</th>
<th>$P_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_1$</td>
<td>$a_1$</td>
<td></td>
<td></td>
<td>$u_1$</td>
<td>$a_1$</td>
<td></td>
<td></td>
<td></td>
<td>$w_1$</td>
<td>$a_1$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$v_2$</td>
<td>$a_2$</td>
<td>$b_2$</td>
<td></td>
<td>$u_2$</td>
<td>$a_2$</td>
<td>$b_2$</td>
<td></td>
<td></td>
<td>$w_2$</td>
<td>$a_2$</td>
<td>$b_2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$v_3$</td>
<td>$a_3$</td>
<td></td>
<td></td>
<td>$u_3$</td>
<td>$a_3$</td>
<td></td>
<td></td>
<td></td>
<td>$w_3$</td>
<td>$a_3$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$v_4$</td>
<td></td>
<td>$b_1$</td>
<td></td>
<td>$u_4$</td>
<td>$b_1$</td>
<td></td>
<td></td>
<td>$d_1$</td>
<td>$w_4$</td>
<td>$b_1$</td>
<td></td>
<td></td>
<td>$d_1$</td>
</tr>
<tr>
<td>$v_5$</td>
<td>$b_3$</td>
<td>$c_3$</td>
<td></td>
<td>$u_5$</td>
<td>$b_3$</td>
<td>$c_3$</td>
<td></td>
<td></td>
<td>$w_5$</td>
<td>$b_3$</td>
<td>$c_3$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$v_6$</td>
<td>$c_1$</td>
<td>$u_6$</td>
<td></td>
<td></td>
<td>$d_2$</td>
<td></td>
<td></td>
<td>$w_6$</td>
<td></td>
<td>$c_1$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$v_7$</td>
<td></td>
<td>$c_2$</td>
<td>$u_7$</td>
<td></td>
<td></td>
<td></td>
<td>$d_3$</td>
<td>$w_7$</td>
<td></td>
<td></td>
<td>$c_2$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Splice.

<table>
<thead>
<tr>
<th>Node</th>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$P_1$</td>
<td>$P_2$</td>
<td>$P_3$</td>
</tr>
<tr>
<td>$v_1$</td>
<td>$a_1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$v_2$</td>
<td>$a_2$</td>
<td>$b_2$</td>
<td></td>
</tr>
<tr>
<td>$v_3$</td>
<td>$a_3$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$v_4$</td>
<td></td>
<td>$b_1$</td>
<td></td>
</tr>
<tr>
<td>$v_5$</td>
<td>$b_3$</td>
<td>$c_3$</td>
<td></td>
</tr>
<tr>
<td>$v_6$</td>
<td>$c_1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$v_7$</td>
<td></td>
<td>$c_2$</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5: Induced graph of a splice.

3.1 Description of the algorithm

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

Phase I (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 Figure 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 (Figure 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 Figure 6 (a and b2) for an example), and adding an edge to a node already on the path (step 2, see Figure 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.

Treatment of undirected graphs is practically the same, differing only in that ExpandPath for undirected

![Figure 6: Phase I example.](image-url)
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 non-cyclic. When adding an additional node (steps 1, 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 there exists a significant difference between directed and undirected graphs, except for code hidden in computing the number of paths pairs.

### Algorithm 1: Frequent paths - Phase I

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

### 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 $\tau$ with $G(\tau) = (P_1, P_2)$, s.t. $\Omega(\tau)$ is connected and $\rho(\Omega(\tau)) = 2$, add tuple-set $\tau$ 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 $\tau$ to $L_2$.
5. Output graphs $\{\Omega(\tau) \mid \tau \in L_2\}$.

### Algorithm 2: Frequent path pairs - Phase II

Phase II (see Algorithm 2) constructs the frequent graphs with path number 2, by combining one-path graphs. The non-trivial 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. Figure 7 shows (in terms of the lexicographic order we defined earlier) how several different 2-path graphs are constructed from two paths.

### Phase III (see Algorithm 3)

Phase III constructs the frequent graphs with path number $n$ from graphs with path number $n-1$. The non-trivial 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 $\tau_i$ are represented in sorted order of the paths in $G(\tau_i)$, where each path is
Figure 8 demonstrates merging two 2-path graphs that are not maximal, i.e. contained in larger frequent graphs. 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 I and II.

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

**Algorithm 3: Frequent graphs - Phase III**

1. Set \( n = 3 \), \( C_n = \emptyset \), \( L_n = \emptyset \).
2. For every pair \( \tau_1, \tau_2 \) 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 \( \tau \) to \( C_n \).
   b. For every composition tuple-set \( S \in L_2 \) if \( \Omega(\text{Splice}(\tau, S)) \) is connected and has path number \( n \), add \( \text{Splice}(\tau, S) \) 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 \( \tau \) 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.

**Notation:**
- \( L_n \): set of composition tuple-sets of width \( n \).
- \( C_n \): a candidate set for these compositions.

**Proof outline:**

1. The fact that non-isomorphic 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.

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

**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 I. Denote the isomorphisms of \( P_1, P_2 \) by \( P'_1, P'_2 \), respectively. During phase II, all possible consistent composition tuple-sets \( \tau \) with \( G(\tau) = (P'_1, P'_2) \) are constructed, including the composition \( \tau \) 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 \( \tau \) is also \( P \)-minimal, and thus not pruned from \( C_2 \) at step 3. Since \( G \) is frequent, \( \tau \) is stored in \( L_2 \) in step 4, and \( G \) is output at step 5.

**Theorem 5** Phase III 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
The pattern, and inherent to apriori-like algorithms. This complexity is exponential in the size of the iteration by using bijective sum and splice to form $G$

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 as well. While the 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 non-dense graphs the algorithm is still reasonable for large graphs.

### 4.1 Experimental setting

The experimental environment is a Sun Ultra-30 workstation running at 247 MHz and 128 MB of main memory. The real XML file we used is a portion of the “movies” database. XML elements are treated as edges. This set of experiments used also two databases, one synthetic, and a real-life social network composed of electronic mails. The database records emails over a period of a week among users of the BGU email system. Source, destination and size of the message were recorded. The message size is used as an approximate "label" on the edge.

### 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 dataset of transactions $D$ is:

$$S(I) = \frac{|\{t \mid t \in D, \langle i_1, \ldots, i_k \rangle \geq t\}|}{|D|}$$

(2)

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 non-trivial 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. Figure 9 shows a database...
The only non-trivial provably admissible measure we could find for the “single-graph”-setting is defined as follows [33]. 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 sub-graphs $A_i$ 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 [33] 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 in the size of $I_G$, since the independent set problem is NP-hard. However, for real-life cases of sparse database graph 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 [15] for details) as a user usually does not care about a precise support value. See further discussion on computing the MIS measure in Section 5.

### 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. The same admissible MIS support measure was used for all 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).

### 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; (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 #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 dataset. 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 names of directors, since these are present only as tags in the XML database. 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 runtime 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 number of candidate and frequent patterns generated by both algorithms for various support values on two subsets (5000 and 2000 nodes) of a Ben-Gurion university e-mail traffic database. The entire database is large (over 50000 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 8 shows numbers of candidates and frequent patterns generated by both algorithms for various support values on random graphs with 3000 nodes, 4000 edges and different numbers of labels: 30, 40 and 50. These results show that our algorithm produces fewer candidate patterns that FSG and therefore performs fewer support computations.

Figure 10 shows numbers of candidates generated by both algorithms for various support values on random graphs with 3000 nodes, 4000 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 pat-
Table 6: Movie DB: support vs. frequent patterns

<table>
<thead>
<tr>
<th>#1</th>
<th>12656</th>
<th>13878</th>
<th>112</th>
<th>FP</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>15</th>
<th>16</th>
<th>16</th>
<th>18</th>
<th>21</th>
<th>22</th>
</tr>
</thead>
<tbody>
<tr>
<td>#2</td>
<td>8337</td>
<td>9416</td>
<td>25</td>
<td>FP</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>#3</td>
<td>7027</td>
<td>7851</td>
<td>22</td>
<td>FP</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>11</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>#4</td>
<td>4730</td>
<td>4813</td>
<td>90</td>
<td>FP</td>
<td>5</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>11</td>
<td>11</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>16</td>
</tr>
<tr>
<td>#5</td>
<td>2757</td>
<td>2794</td>
<td>76</td>
<td>FP</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>#6</td>
<td>1293</td>
<td>1292</td>
<td>91</td>
<td>FP</td>
<td>21</td>
<td>32</td>
<td>34</td>
<td>46</td>
<td>79</td>
<td>79</td>
<td>84</td>
<td>84</td>
<td>86</td>
<td>86</td>
</tr>
</tbody>
</table>

Table 7: BGU e-mail database results

<table>
<thead>
<tr>
<th>5000 nodes</th>
<th>2000 nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>C</td>
</tr>
<tr>
<td>1%</td>
<td>54</td>
</tr>
<tr>
<td>0.9%</td>
<td>54</td>
</tr>
<tr>
<td>0.8%</td>
<td>65</td>
</tr>
<tr>
<td>0.7%</td>
<td>65</td>
</tr>
<tr>
<td>0.6%</td>
<td>65</td>
</tr>
<tr>
<td>0.5%</td>
<td>65</td>
</tr>
<tr>
<td>0.4%</td>
<td>77</td>
</tr>
<tr>
<td>0.3%</td>
<td>77</td>
</tr>
<tr>
<td>0.2%</td>
<td>119</td>
</tr>
<tr>
<td>0.1%</td>
<td>230</td>
</tr>
</tbody>
</table>

Table 8: Random graph with 3000 nodes and 4000 edges

Figure 10: Random graph with 1000 nodes and 2000 edges. 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.

[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 tree-like patterns in semi-structured documents, represented in the standard OEM model. Although this algorithm searches only for tree-like patterns, it can also handle patterns containing cycles by transforming them into trees. 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, Nijssen, Muntz and Kok handle the problem of tree mining in a wider sense in [28].

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 item-set mining in structured databases and the Apriori algorithm and its variations [1]. For conciseness,
Other works which use the BFS approach are [19, 16, 17, 6]. The second approach, called gSpan [40], grows 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 sub-graphs, are present in 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.

Regarding the greedy approach, two categories of algorithms were mentioned in the introduction: transaction graphs and single graph settings. To-date, most work was for the transaction graph setting, with algorithms are 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 \( K \). [18] uses vertices. An algorithm by Karamochi and Karpis [20] uses edges as the main building block, and was extended and improved in [21] by adding several clever heuristics which make mining and support computation more efficient. This latter version, called FSG, is currently one of the best known and often compared to version of 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 sub-graphs, are present in 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.

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 on-line documents, which 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 apriori-TID technique.

**Aknowledgements**

Partially supported by the KITE consortium under contract to the Israeli Ministry of Trade and Indus-
References


[4] Y. Chi, S. Nijssen, R. R. Muntz, and J. N. Kok, Frequent subtree mining: an overview, Fundamenta Informaticae, Special Issue on Graph and Tree Mining, 2005


[15] Production Management. We 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.


[22] M. Kuramochi and G. Karypis, Finding Frequent Patterns in a Large Sparse Graph Proceedings 2004 SIAM Data Mining Conference, Orlando, Florida, 2004


N. Vanetik and E. Gudes, Mining Frequent Labeled and Partially Labeled Graph Patterns, Proceedings of ICDE 2004: 91-102, Boston, MA 2004.


X. Yan and J. Han, gSpan: Graph-Based Substructure Pattern Mining, Proceedings of ICDM 2002: 721-724.

X. Yan and J. Han, CloseGraph: mining closed frequent graph patterns, Proceedings of KDD03, Washington, 2003.


Solomon Eyal Shimony received the BSc degree in Electrical Engineering from the Technion in 1982, a PhD in Computer Science from Brown University in 1991, after which he joined the department of Computer Science at Ben-Gurion University. At present, he is a deputy head of CS department at BGU, chair of the Paul Ivanier Center for Robotics and Production Management, and an associate editor of IEEE Systems, Man, and Cybernetics, section B. His research interests are artificial intelligence, probabilistic reasoning, knowledge discovery in databases, robotics, flexible computation, and spatial data models.

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