Computing frequent patterns from semi-structured data

Natalia Vanetik, Ehud Gudes and Eyal S. Shimony

Department of Computer Science,
Ben-Gurion University of the Negev

Talk outline

Part 1 Introduction
  1.1 Motivation and background
  1.2 Overview of the graph mining algorithm

Part 2 Details
  2.1 The support measure
  2.2 Combining graphs
  2.3 Details of the mining algorithm

Part 3 Experimental evaluation

Part 4 Conclusions and future work
What Graphs are good for?

- Most of existing data mining algorithms are based on transaction representation, i.e., sets of items.
- Datasets with structures, layers, hierarchy and/or geometry often do not fit well in this transaction setting. For e.g.
  - Numerical simulations
  - 3D protein structures
  - Chemical Compounds
  - Generic XML files.

Graph Based Data Mining

- Graph Mining then essentially is the problem of discovering repetitive subgraphs occurring in the input graphs.

- Motivation:
  - finding subgraphs capable of compressing the data by abstracting instances of the substructures.
  - Identifying conceptually interesting patterns
Overview

Semi-structured data is any data that can be modeled as a labeled graph. For example, XML and HTML data, user access patterns.

Frequent patterns discovered from semi-structured data are useful for:

- Improving database design (A. Deutsch, M. Fernandez, D. Suciu “Storing Semistructured Data with STORED”, SIGMOD’99)
- Efficient indexing (R. Goldman, J. Widom “DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases”, VLDB’97)
- User preference based applications
- User behavior predictions
- Database storage and archival

Rule-based patterns
Patterns of form $A_1,A_2,...,A_n \Rightarrow B$ where $A_1,...,A_n,B$ are atomic values.
Example: “diapers ⇒ beer”.

Topology-based patterns
Patterns that have structure in addition to atomic values
Example: graph patterns

\[
\begin{aligned}
\text{university} & \quad \text{staff} \\
\text{staff} & \quad \text{research} \\
\text{research} & \quad \text{publications} \quad \text{research interests}
\end{aligned}
\]
Overview

**Definition** Transaction database is a set of records where each record contains results of a single transaction.

**Example:** Supermarket database where each purchase is a transaction.

An item-set $X$ in relational model is a set of tuples $(f_1,v_1),\ldots,(f_n,v_n)$ where $f_i$ are the names of fields and $v_i$ are values. $n$ is a size of an item-set.

A transaction $T$ supports $X$ if the value of $f_i$ equals $v_i$ for any $i$.

Let user-defined support threshold be $s\%$. An item-set $X$ is frequent if $s\%$ or more transactions in the database support $X$.

Apriori algorithm

**Apriori principle:** If $X$ is a frequent item-set of size $n$ then all items $X'$ contained in $X$ are frequent as well.

**Apriori algorithm:** each item-set $X$ of size $n$ is first generated from two frequent item-sets $X_1$ and $X_2$ of size $(n-1)$ and then its frequency is evaluated by a pass over the database.
Background

**Existing algorithms**

- Simple path patterns (Chen, Park, Yu 98)
- Generalized path patterns (Nanopoulos, Manolopoulos 01)
- Simple tree patterns (Lin, Liu, Zhang, Zhou 98)
- Tree-like patterns (Wang, Huiqing, Liu 98)
- “Naïve” graph patterns (Kuramochi, Karypis 01)

---

**FSG Algorithm**

[M. Kuramochi and G. Karypis. *Frequent subgraph discovery.*]

- Incremental and breadth-first fashion in the size of frequent subgraphs (like Apriori for frequent itemsets)
- Counting of single and double edge subgraphs
- For finding frequent subgraphs from size $k$ ($k > 2$).
  - Candidates generation- all possible joining of two graphs from size $k-1$ which share common kernel subgraph from size $k-2$.
  - Candidate pruning- a necessary condition of candidate to be frequent is that each of its subgraphs is frequent.
  - Frequency counting - Check if a candidate subgraph appear at least minSup (minimum support) times.
  - Repeat the steps for $k=k+1$
GSpan Algorithm

[X. Yan and J. Han. GSpan: Graph-based substructure pattern mining.]

- Adopts a pattern-growth by growing patterns from a single graph directly.
- The algorithm maps each subgraph to a unique label.
- By using these labels, a Tree Search-Space (TSS) hierarchy is constructed over all possible subgraphs.
- A subgraph from size $k$ is kept in node at depth $k$ in TSS.
- An in-order search over the TSS lets discover all frequent subgraphs.
  - Pruning - If a node in TSS holds infrequent subgraph then its sub-tree in TSS is pruned.

Overview

**Model:** A semistructured database is viewed as a (labeled) graph. A pattern is any connected subgraph of a database graph.

**Goal:** Find all frequent connected subgraphs of a database graph.

**Related problem:** What if there are no transactions in the database (when it is a single graph, such as XML file, Web etc.)? How do we count pattern instances? What is support of a graph pattern?

**Most existing algorithms use Transactions based databases!**

**We assume a single graph database!**
Overview (contd.)

Most existing algorithms use Apriori-based approach.

The basic building block is either a tree (for tree mining only) or an edge.

**Our approach:** use edge-disjoint paths as building blocks.

**Result:** faster convergence of the algorithm

---

**Path Facts**

**Definition** Path number $p(G)$ of a graph is the minimal number of edge-disjoint paths that cover all edges in the graph. A collection of $p(G)$ paths that cover all edges is called a minimal path cover.

Graph $G$ is **Eulerian** if it can be covered by a single cyclic path (in this case, $p(G)=1$).

For a non-Eulerian connected graph $G$, the following is true:

$$p(G) = \frac{1}{2} \left| \left\{ v \in V \mid d(v) \text{ is odd} \right\} \right|$$  

(for an undirected graph)

$$p(G) = \frac{\sum_{v \in V} \left| d^+(v) - d^-(v) \right|}{2}$$  

(for a directed graph)
Path Facts

**Definition** Graph $G' = G \backslash P$ where $P$ is an edge-disjoint path denotes the graph obtained by removing from $G$ all edges of $P$ followed by removing of all trivial sub-graphs.

**Claim 1:** Let $P$ be any path from minimal path cover of a connected graph $G$. Then $p(G \backslash P) = p(G) - 1$.

**Claim 2:** In any path cover of connected graph $G$ there are at least two paths $P_1, P_2$ such that $G \backslash P_1$ and $G \backslash P_2$ are connected.

We also define an order $\preceq$ on paths in order to represent a path decomposition of a graph in a unique way. We only store decompositions that are minimal with respect to this order (denoted by $P$-minimal).

Three phases of mining algorithm

- Phase #1 finds all frequent graph patterns with path number 1
- Phase #2 finds all frequent graph patterns with path number 2 by “joining” pairs of patterns found in phase #1
- Phase #3 finds all frequent graph patterns with path number $n \geq 3$ by “joining” pairs of patterns with path number $(n-1)$.
Support issue

Definition a support measure $S$ is admissible if for any pattern $P$ and any sub-pattern $Q \subset P$, $S(Q) \geq S(P)$.

Problem: the number of appearances of the graph pattern in the database graph is not an admissible support measure.

Graph $A$ appears 3 times in the database graph, while graph $B \subset A$ appears only once.

Support issue

Definition An instance graph $I(P)$ of pattern $P$ in database graph $D$ is a graph $G = (V,E)$ where $V = \{g \subset G, g \approx P\}$ and $E = \{(g,h), g,h \in V \text{ and } E(g) \cap E(h) \neq \emptyset\}$.

Operations on instance graph:

- **clique contraction**
  replacing a clique $C$ by a single node $c$ such that only the nodes that were adjacent to each node of $C$ may become adjacent to $c$

- **node expansion**
  replacing an existing node $v$ by a new subgraph whose nodes may or may not be adjacent to the nodes adjacent to $v$

- **node addition**
  adding a new node to the graph and arbitrary edges between the new node and the old ones

- **edge removal**
Example of operations on instance graph

- clique contraction
- vertex addition
- edge removal
- vertex expansion

The main result

Theorem  A support measure $S$ is an admissible support measure if it is non-decreasing on instance graph $I(P)$ of every pattern $P$ under clique contraction, node expansion, node addition and edge removal.

Note  we proved a stronger result: support measure is admissible if and only if it is non-decreasing on $I(P)$ under these operations.
Example of support measure

*Admissible* support measure:

Maximum independent set size of instance graph

Number of edges in the database graph

Motivation:

We are interested in *typical* structure, i.e. structures created by many users. A single complex structure that has many references is less interesting for us.

Most common support measures are covered by this definition, including the standard one for transaction databases.

---

Graph composition - composition relation

**Definition** A *composition relation* \( C(P_1, \ldots, P_n) \) on paths \( P_1, \ldots, P_n \) of graph \( G \) is a table with nodes of \( G \) as rows and paths as columns such that \( C[i,j] \neq \bot \) iff \( i \)-th node of \( G \) is also a node of path \( P_j \).

**Example:** \( C(P_1, P_2, P_3) \) as a table

<table>
<thead>
<tr>
<th>Node</th>
<th>( P_1 )</th>
<th>( P_2 )</th>
<th>( P_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1</td>
<td>a1</td>
<td>( \bot )</td>
<td>( \bot )</td>
</tr>
<tr>
<td>v2</td>
<td>a2</td>
<td>b2</td>
<td>( \bot )</td>
</tr>
<tr>
<td>v3</td>
<td>a3</td>
<td>( \bot )</td>
<td>( \bot )</td>
</tr>
<tr>
<td>v4</td>
<td>( \bot )</td>
<td>b1</td>
<td>( \bot )</td>
</tr>
<tr>
<td>v5</td>
<td>( \bot )</td>
<td>b3</td>
<td>c3</td>
</tr>
<tr>
<td>v6</td>
<td>( \bot )</td>
<td>( \bot )</td>
<td>c1</td>
</tr>
<tr>
<td>v7</td>
<td>( \bot )</td>
<td>( \bot )</td>
<td>c2</td>
</tr>
</tbody>
</table>
Graph composition

By treating table rows as graph nodes and defining edges \((i,j)\) whenever two nodes of a path \(P_k\), appearing in rows \(i\) and \(j\), have an edge between them, we can construct a graph corresponding to composition relation \(C(P_1,\ldots,P_n)\).

**Notation:** graph composition (realization) \(\Omega(C)\) of \(C(P_1,\ldots,P_n)\).

**Example:** Graph composition \(\Omega(C(P_1,P_2,P_3))\).

Subtraction from Composition Relation

**Definition** Subtraction of a path \(P_i\) from composition relation consists of:

a) eliminating the \(i\)-th column from the table;
b) removal of all rows containing only null values.

**Example:** \(C(P_1,P_2,P_3)\) after subtraction of \(P_3\), denoted by \(C \setminus P_3\) or \(C_{\{1,2\}}\).
**Bijective sum**

**Definition** A bijective sum $BS(C_1, C_2, I_1, I_2)$ of composition relations $C_1$ and $C_2$, where $I_1, I_2$ are sets of indices and $C_1|_{I_1} = C_2|_{I_2}$, is a composition relation obtained by adding all columns of $C_2$ corresponding to paths that are not in $C_1$, to the table of $C_1$.

**Example:** Bijective sum of $C_1$ and $C_2$ on common paths $P_1$ and $P_2$.

<table>
<thead>
<tr>
<th>C_1</th>
<th>C_2</th>
<th>C_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1</td>
<td>a1</td>
<td>v1</td>
</tr>
<tr>
<td>v2</td>
<td>a2</td>
<td>v2</td>
</tr>
<tr>
<td>v3</td>
<td>a3</td>
<td>v3</td>
</tr>
<tr>
<td>v4</td>
<td>b1</td>
<td>v4</td>
</tr>
<tr>
<td>v5</td>
<td>b3</td>
<td>c3</td>
</tr>
<tr>
<td>v6</td>
<td>c1</td>
<td>v6</td>
</tr>
<tr>
<td>v7</td>
<td>c2</td>
<td>v7</td>
</tr>
<tr>
<td>v8</td>
<td>d2</td>
<td>v8</td>
</tr>
<tr>
<td>v9</td>
<td>d5</td>
<td>v9</td>
</tr>
</tbody>
</table>

**Graph composition of a bijective sum**

[Diagram of graph compositions]
Splice

**Definition** A splice $\oplus_{ij}$ of two composition relations $C_1(P_1,\ldots,P_n)$ and $C_2(P_i,P_j)$, is a composition relation that turns every node common to $P_i$ and $P_j$ in $C_2$, into the node common to $P_i$ and $P_j$ in $C_1$ as well.

**Example:** $C_3 = C_1(P_1,P_2,P_3) \oplus_{2,3} C_2(P_2,P_3)$.
Phase #1 – overview

Definition: A node v in graph G is balanced if degree of v is even (for undirected graphs). A node is unbalanced if it is not balanced.

Phase #1 constructs frequent paths by adding one edge at a time.

If the path is cyclic (i.e., a (not necessarily simple) cycle), we can add edge anywhere (providing the labels match):
1. between two existing nodes,
2. between existing and new node.

If the path is not cyclic, we can add edge between pair of nodes one of which is unbalanced:
1. between two existing unbalanced nodes,
2. between existing unbalanced and existing balanced node,
3. between existing unbalanced node and a new node.

Phase #1 – Path generation

Algorithm: Phase #1

1. Find all frequent edges and add them to L_{1,1}. Set k←2.
2. Set C_{1,k}←∅, L_{1,k}←∅.
3. For every path P∈L_{1,k-1} and every edge e=(v,u)∈L_{1,1} do:
   a. Let X be all nodes of P if P is cyclic and all unbalanced nodes of P if P is non-cyclic.
   b. For every x∈X such that x≈v, add Q=(V(P)∪{u}, E(P)∪{x,u}) to C_{1,k} if p(Q)=1.
   c. For every x∈X such that x≈u, add Q=(V(P)∪{v}, E(P)∪{(v, x)}) to C_{1,k} if p(Q)=1.
   d. For every x,y∈X such that x≈v, x≈u and (x,y)∉E(P), add Q=(V(P), E(P)∪{(x, y)}) to C_{1,k} if p(Q)=1.
4. Compute frequency of all paths from C_{1,k} and add the frequent ones to L_{1,k}.
5. If L_{1,k}=∅, stop. Otherwise, set k←k+1 and go to step 2.
Phase #1 – Example

Algorithm: Phase #2

1. Let \( L_1 \) be the set of all frequent paths.
   Set \( C_2 \leftarrow \emptyset \), \( L_2 \leftarrow \emptyset \).
2. For every pair \( P_1, P_2 \in L_1 \) and every possible label-preserving composition relation \( C \) on \( P_1 \) and \( P_2 \) do:
   a. If \( p(\Omega(<P_1, P_2>, C)) = 2 \), add \( <P_1, P_2, C> \) to \( C_2 \).
3. Remove all tuples producing non P-minimal graphs from \( C_2 \).
4. For every \( t \in C_2 \) if \( \Omega(t) \) is frequent, add it to \( L_2 \).

Phase #2 – Path pairs generation
Phase #2 - Example

Join of two paths produced three graphs with path number 2

Phase #3 – overview

Phase #3:  
**Input** = frequent graphs with path number $k$  
**Output** = frequent graphs with path number $(k+1)$

**The main step:**
1. find a common $(k-1)$-subgraph of two $k$-graphs,
2. if found, join these graphs into $(k+1)$-graph using bijective sum operation,

**Additional step:**
3. for bijective sum $G$ of two graphs and two paths $P$ and $Q$ in which these graphs differ, find all frequent combinations of $P$ and $Q$ in $L_2$, and join them with $G$ using splice operation.
Phase #3 – Graphs with \( p(G) \geq 3 \)

Algorithm: Phase #3

1. Let \( L_2 \) be the set of all frequent path pairs. Set \( k \leftarrow 3 \).
2. Set \( C_k \leftarrow \emptyset \), \( L_k \leftarrow \emptyset \).
3. For every \( t_1, t_2 \in L_{k-1} \) such that \( t_1 = \langle P_1, \ldots, P_{i-1}, P_i, \ldots, P_j, \ldots, P_k, C_1 \rangle \) and \( t_2 = \langle P_1, \ldots, P_j, \ldots, P_{j-1}, P_{j+1}, \ldots, P_k, C_2 \rangle \) do:
   a. Let \( C = BS(C_1, C_2, (k)-i-j, (k)-i-j) \).
   b. Add \( t = \langle P_1, \ldots, P_k, C \rangle \) to \( C_k \) (if \( p(\Omega(t)) = k \)).
   c. For every \( t_3 = \langle P_i, P_j, C_3 \rangle \in L_2 \), add \( t = \langle P_1, \ldots, P_k, C \oplus_i,j C_3 \rangle \) to \( C_k \) (if \( p(\Omega(t)) = k \)).
4. Remove all non P-minimal tuples from \( C_k \).
5. Add every \( t \in C_k \), where \( \Omega(t) \) is frequent, to \( L_k \).
6. If \( L_k = \emptyset \), stop. Otherwise, set \( k \leftarrow k+1 \) and go to step 2.

Proof milestones

**Theorem 1** All frequent graphs with path number 1 are produced by phase 1 of the algorithm.

**Basis:** For every path \( P \) and unbalanced vertex \( v \) of \( P \) there exists a vertex \( u \) such that \( (u,v) \in E(P) \) and \( P \setminus (u,v) \) is a path.

**Theorem 2** All frequent graphs with path number 2 are produced by phase 2 of the algorithm.

**Basis:** Each graph \( G \) with \( p(G) = 2 \) can be expressed as a label-preserving composition relation on two paths from its any P-minimal path decomposition.

**Theorem 3** All frequent graphs with \( p(G) > 2 \) are produced by phase 3.

**Main steps:**
1. There exists two paths \( P, Q \) in minimal path decomposition of \( G \) such that \( G \setminus P \) and \( G \setminus Q \) are connected.
2. \( G \setminus P \) and \( G \setminus Q \) are also frequent and were found earlier.
3. If \( P \) and \( Q \) are disjoint in \( G \), then application of a bijective sum produces \( G \).
4. Otherwise, bijective sum and splice combined produce \( G \).
Complexity

Exponential – as the number of frequent patterns can be exponential on the size of the database

Difficult tasks:
1. Support computation that consists of:
   a. Finding all instances of a frequent pattern in the database.
   b. Computing MIS (maximum independent set size) of an instance graph.

Relatively easy tasks:
1. Candidate set generation:
   polynomial on the size of frequent set from previous iteration,
2. Elimination of isomorphic candidate patterns:
   graph isomorphism computation is at worst exponential on the size of a pattern, not a database.

Phase #3 – complexity

Algorithm: Phase #3

1. Let \( L_2 \) be the set of all frequent path pairs. Set \( k \leftarrow 3 \).
2. Set \( C_k \leftarrow \emptyset \), \( L_k \leftarrow \emptyset \).
3. For every \( t_1, t_2 \in L_k \) such that \( t_1 = \langle P_1, P_2, \ldots, P_i, C_1 \rangle \) and \( t_2 = \langle P_1, P_2, \ldots, P_j, C_2 \rangle \) do:
   a. Let \( C = \text{BS}(C_1, C_2, (k) - i - j, (k) - i - j) \).
   b. Add \( t = \langle P_1, \ldots, P_k, C \rangle \) to \( C_k \) (if \( p(\Omega(t)) = k \)).
   c. For every \( t_3 = \langle P_i, P_j, C_3 \rangle \in L_2 \), add \( t = \langle P_1, \ldots, P_k, C \oplus C_3 \rangle \) to \( C_k \) (if \( p(\Omega(t)) = k \)).
4. Remove all non P-minimal tuples from \( C_k \).
5. Add every \( t \in C_k \), where \( \Omega(t) \) is frequent, to \( L_k \).
6. If \( L_k = \emptyset \), stop. Otherwise, set \( k \leftarrow k + 1 \) and go to step 2.
Complexity (cont.)

*Why is mining in real-life databases easier?*

- real databases tend to be sparse rather than dense,
- real databases tend to have large number of different labels.

*Impact on algorithm’s complexity:*

- the number of database subgraphs isomorphic to a given graph pattern is not exponential,
- the size of instance graph is not exponential,
- instance graphs tend to be very sparse, which makes the task of finding MIS much easier.

*Additional improvements:*

- approximate techniques can be used for MIS computation as user usually does not care for the *exact* support value.

Experiment overview

*Goals of our experiments are:*

- To compare our algorithm with naïve algorithms:
  - **Naive1** – produce *all* graphs and compute their support (B. D. McKay *Isomorph-free exhaustive generation*, J. of Algorithms vol. 26, 1998)
  - **Naive2** – at each iteration, add edge to frequent graphs from previous iteration

- To study algorithm’s behavior on various graph topologies:
  - cliques
  - trees
  - sparse graphs vs dense graphs

- To study the effect of following parameters on the number of frequent patterns found:
  - size of the database
  - number of different labels

- Test algorithm on both synthetic and real-life databases
Experiments setting

Notation: S – support; N – nodes; L – labels; E – edges
FP – frequent patterns
C – candidate patterns; I – isomorphism checks;
SC – support calculations; ALG - algorithm in use.

Experimental results on synthetic data: trees

<table>
<thead>
<tr>
<th>#</th>
<th>N</th>
<th>L</th>
<th>S</th>
<th>FP</th>
<th>ALG</th>
<th>C</th>
<th>I</th>
<th>Sc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>40</td>
<td>4</td>
<td>7%</td>
<td>15</td>
<td>Naive2</td>
<td>100</td>
<td>24</td>
<td>92</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
<td>4</td>
<td>7%</td>
<td>16</td>
<td>Naive2</td>
<td>110</td>
<td>43</td>
<td>102</td>
</tr>
<tr>
<td>3</td>
<td>50</td>
<td>6</td>
<td>3%</td>
<td>37</td>
<td>Naive2</td>
<td>470</td>
<td>82</td>
<td>458</td>
</tr>
<tr>
<td>4</td>
<td>50</td>
<td>8</td>
<td>3%</td>
<td>27</td>
<td>Naive2</td>
<td>202</td>
<td>229</td>
<td>205</td>
</tr>
<tr>
<td>5</td>
<td>60</td>
<td>4</td>
<td>8%</td>
<td>15</td>
<td>Naive2</td>
<td>108</td>
<td>24</td>
<td>95</td>
</tr>
<tr>
<td>6</td>
<td>60</td>
<td>6</td>
<td>8%</td>
<td>44</td>
<td>Naive2</td>
<td>728</td>
<td>203</td>
<td>716</td>
</tr>
<tr>
<td>7</td>
<td>60</td>
<td>8</td>
<td>8%</td>
<td>14</td>
<td>Naive2</td>
<td>103</td>
<td>18</td>
<td>87</td>
</tr>
</tbody>
</table>
## Experimental results on synthetic data: sparse graphs

<table>
<thead>
<tr>
<th>S</th>
<th>N</th>
<th>E</th>
<th>L</th>
<th>S</th>
<th>FP</th>
<th>ALG</th>
<th>C</th>
<th>I</th>
<th>SC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>40</td>
<td>50</td>
<td>4</td>
<td>7%</td>
<td>14</td>
<td>Naive2</td>
<td>60</td>
<td>33</td>
<td>52</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Apriori</td>
<td>49</td>
<td>55</td>
<td>42</td>
</tr>
<tr>
<td>2</td>
<td>40</td>
<td>50</td>
<td>6</td>
<td>3%</td>
<td>17</td>
<td>Naive2</td>
<td>84</td>
<td>48</td>
<td>76</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Apriori</td>
<td>59</td>
<td>70</td>
<td>54</td>
</tr>
<tr>
<td>3</td>
<td>50</td>
<td>60</td>
<td>6</td>
<td>3%</td>
<td>28</td>
<td>Naive2</td>
<td>355</td>
<td>74</td>
<td>343</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Apriori</td>
<td>117</td>
<td>185</td>
<td>143</td>
</tr>
<tr>
<td>4</td>
<td>60</td>
<td>80</td>
<td>4</td>
<td>4%</td>
<td>16</td>
<td>Naive2</td>
<td>101</td>
<td>31</td>
<td>93</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Apriori</td>
<td>56</td>
<td>58</td>
<td>56</td>
</tr>
<tr>
<td>5</td>
<td>60</td>
<td>80</td>
<td>6</td>
<td>3%</td>
<td>27</td>
<td>Naive2</td>
<td>265</td>
<td>86</td>
<td>253</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Apriori</td>
<td>120</td>
<td>102</td>
<td>110</td>
</tr>
<tr>
<td>6</td>
<td>70</td>
<td>90</td>
<td>8</td>
<td>3%</td>
<td>27</td>
<td>Naive2</td>
<td>252</td>
<td>77</td>
<td>236</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Apriori</td>
<td>126</td>
<td>98</td>
<td>119</td>
</tr>
<tr>
<td>7</td>
<td>80</td>
<td>100</td>
<td>8</td>
<td>3%</td>
<td>32</td>
<td>Naive2</td>
<td>460</td>
<td>74</td>
<td>387</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Apriori</td>
<td>149</td>
<td>127</td>
<td>141</td>
</tr>
</tbody>
</table>

## Subsets of Movie database used in experiments

<table>
<thead>
<tr>
<th>Data set #</th>
<th>Nodes</th>
<th>Edges</th>
<th>Labels</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12656</td>
<td>13878</td>
<td>112</td>
</tr>
<tr>
<td>2</td>
<td>8337</td>
<td>9416</td>
<td>25</td>
</tr>
<tr>
<td>3</td>
<td>7027</td>
<td>7851</td>
<td>22</td>
</tr>
<tr>
<td>4</td>
<td>4730</td>
<td>4813</td>
<td>90</td>
</tr>
<tr>
<td>5</td>
<td>2757</td>
<td>2794</td>
<td>76</td>
</tr>
<tr>
<td>6</td>
<td>1293</td>
<td>1292</td>
<td>91</td>
</tr>
</tbody>
</table>
Experimental results on subsets of Movie database

<table>
<thead>
<tr>
<th>Dataset</th>
<th>#1</th>
<th>#2</th>
<th>#3</th>
<th>#4</th>
<th>#5</th>
<th>#6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of frequent patterns</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>90%</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>20%</td>
</tr>
<tr>
<td>80%</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>9</td>
<td>10%</td>
</tr>
<tr>
<td>70%</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>9%</td>
</tr>
<tr>
<td>60%</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>9</td>
<td>8%</td>
</tr>
<tr>
<td>50%</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>21</td>
<td>8%</td>
</tr>
<tr>
<td>40%</td>
<td>6</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td>32</td>
<td>6%</td>
</tr>
<tr>
<td>30%</td>
<td>7</td>
<td>5</td>
<td>8</td>
<td>5</td>
<td>34</td>
<td>5%</td>
</tr>
</tbody>
</table>

Comparison

Our algorithm vs naive ones
- Naive1 algorithm does not work on graphs with ≥ 10 nodes
- Our algorithm produces less candidate patterns and therefore performs less support computations than Naive2 algorithm.

Trees vs sparse graphs
- Support computation is easier for trees
- Less candidate patterns are generated for trees

Synthetic vs real-life data
- Synthetic graphs are not very regular. When increasing number of labels, the chance of finding non-trivial frequent graph patterns decreases drastically.
  Large real-life graph databases are highly regular and contain complex frequent graph patterns.
Experimental results (on synthetic data)

- Frequency of patterns vs. support
- Time vs. support

Series 1, Series 2, Series 3
Pattern examples in Movies database

G2: 
- movie
   - title
   - film
   - director

G7: 
- movie
  - director
  - movie

G14: 
- movie
  - actor
  - movie
  - director

Conclusion

- An Apriori-like algorithm for mining graph patterns that uses edge-disjoint paths as building blocks has been constructed.
- A problem of defining support measure for semi-structured data was addressed.
- An experimental analysis of the algorithm was conducted.
Future work

- Usage of building blocks other than edge disjoint paths, such as trees.
- Using Apriori-TID technique at the advanced stages of the search.
- Treat patterns that have high degree of resemblance, such as bisimular patterns, as representatives of their equivalency classes and generate representatives of each class instead of the full search.
- Find additional examples of admissible support measures.
- Take into account topological properties of a database graph while computing support.
- Compare with GSPAN algorithm using our Support measure.