Mining graph patterns

Davide Mottin, Anton Tsitsulin

Hasso Plattner Institute

Graph Mining course Winter Semester 2017
Lecture road

Subgraph mining

Mining Frequent Subgraphs
Small graphs

Mining Frequent Subgraphs
Large graphs

Not covered this year
Why Frequent patterns

- **Frequent pattern**: a structure (a set of items, subsequences, substructures, etc.) that occurs frequently in a data set

- **Motivation**: Finding inherent regularities in data
  - What products were often purchased together?
  - What are the subsequent purchases after buying a PC?
  - What sequences of DNA are sensitive to this new drug?
  - Which topics are in a collection of documents?
The Apriori principle

- Also called Downward closure Property
- All subsets of a frequent pattern must also be frequent
  - Because any item that contains $X$ must also contain a subset of $X$.

If we have already verified that $X$ is infrequent, there is no need to count $X$’s supersets because they MUST be infrequent too.
Graph Pattern Mining

- **Frequent subgraphs**
  - A (sub)graph is *frequent* if its support (occurrence frequency) in a given dataset is no less than a minimum support threshold

- **Applications of graph pattern mining:**
  - Mining biochemical structures
  - Program control flow analysis
  - Mining XML structures or Web communities
  - Building blocks for graph classification, clustering, compression, comparison, and correlation analysis
Frequent Subgraph Mining

Problem
Find all subgraphs of G that appear at least $\sigma$ times

Suppose $\sigma = 2$, the frequent subgraphs are (only edge labels)
- a, b, c
- a-a, a-c, b-c, c-c
- a-c-a ...

Exponential number of patterns!!!
How to mine frequent subgraphs?

- **Apriori-based approaches**
  - Start with small-size subgraphs and proceeds in a bottom-up manner
  - Join two patterns to create bigger size patterns (through *Apriori principle*)
  - Several approaches
    - AGM/AcGM [Inokuchi et al., PAKDD’00]
    - FSG [Kuramochi et al., ICDM’01]
    - PATH# [Vanetik et al., ICDM’02/ICDM’04]

- **Pattern-growth approaches**
  - Extends existing frequent graphs by adding one edge
  - Several approaches:
    - MoFa [Borgelt et al., ICDM’02]
    - gSpan [Yan et al., ICDM’02]
    - Gaston [Nijssen et al., KDD’04]
    - FFSM [Huan et al., ICDM’03]
    - SPIN [Huan et al., KDD’04]
Apriori-Based Approach

Join operation among graphs is extremely expensive

Join 2 patterns from the previous level

Problems?
Pattern Growth Method

Generate patterns expanding existing ones

\[ G \rightarrow G_1 \rightarrow G_2 \rightarrow \ldots \rightarrow G_n \rightarrow (k+1)\text{-graph} \rightarrow (k+2)\text{-graph} \]

Problems?
duplicate graphs
Other Mining Functions

- **Maximal frequent subgraph mining**
  - A subgraph is maximal, if none of its super-graphs is frequent

- **Closed frequent subgraph mining**
  - A frequent subgraph is closed, if all its supergraphs have a (strictly) smaller frequency
  - Algorithms: CloseGraph, SPIN, MARGIN

- **Significant subgraph mining**
  - Mining subgraphs using some significant test (e.g., G-test, p-value)
  - Algorithms: gBoost, gPLS, Leap, GraphSig

- **Representative orthogonal graphs mining**
  - Mining subgraphs with bounded similarity and overlap with respect to other patterns
  - Algorithms: ORIGAMI
Lecture road

Subgraph mining

Mining Frequent Subgraphs
Small graphs

Mining Frequent Subgraphs
Large graphs
Graph Pattern Mining - Set of graphs

Frequent subgraph
Min support = 3/4

Support: frequency of a subgraph appearing in a set of graphs

Apriori principle (for graphs):
If a graph is frequent, all of its subgraphs are frequent
Subgraph Mining problem

- **Support:** Given a set of labeled graphs $D = \{G_1, G_2, ..., G_n\}$, $G_i = \langle V_i, E_i, \ell_i \rangle$ and a subgraph $G$, the supporting set of $G$ is $D_G = \{G_i | G \subseteq G_i, G_i \in D\}$, where $G \subseteq G_i$ indicates that $G$ is subgraph isomorphic to $G_i$. The support is defined as $\sigma(G) = \frac{|D_G|}{|D|}$

- **Input**
  - Set of labeled-graphs $D = \{G_1, G_2, ..., G_n\}$, $G_i = \langle V_i, E_i, \ell_i \rangle$
  - Minimum support $\text{min\_sup}$

- **Output:**
  - A subgraph $G$ is frequent if $\sigma(G) \geq \text{min\_sup}$
  - Each subgraph is connected.

Support is anti-monotone: for all $G' \subseteq G$, $\sigma(G') \geq \sigma(G)$

On undirected, labeled set of graphs
Finding Frequent Subgraphs: Input and Output

- **Input**
  - Set of labeled graphs (graph database)
  - Minimum support threshold min_sup.

- **Output**
  - Frequent subgraphs that satisfy the minimum support constraint.
  - Each frequent subgraph is connected.
Mining approaches: agenda

- Apriori-based approaches:
  - FSG

- Pattern-growth approaches:
  - gSpan

- Greedy approach:
  - Subdue
FSG Algorithm

**Notation:** k-subgraph is a subgraph with k edges.

**Init:** Scan the transactions to find $F_1$, the set of all frequent 1-subgraphs and 2-subgraphs, together with their counts;

For ($k=3; \ F_{k-1} \neq \emptyset ; \ k++)$

1. **Candidate Generation** - $C_k$, the set of candidate k-subgraphs, from $F_{k-1}$, the set of frequent (k-1)-subgraphs;
2. **Candidates pruning** - a necessary condition of candidate to be frequent is that each of its (k-1)-subgraphs is frequent.
3. **Frequency counting** - Scan the graph database to count the occurrences of subgraphs in $C_k$;
4. $F_k = \{ \ c \in C_k \mid c \text{ has counts no less than } min\_sup \}$
5. Return $F_1 \cup F_2 \cup \ldots \cup F_k$ (= $F$)

Simple operations?

- **Candidate generation**
  - To determine two candidates for joining, we need to check for graph isomorphism

- **Candidate pruning**
  - To check downward closure property, we need graph isomorphism

- **Frequency counting**
  - To check containment of a frequent subgraph, we need subgraph isomorphism

Recall that subgraph isomorphism is NP-complete!!!
Candidates generation (join) based on core detection - Issues

Same core different $k+1$ patterns

By Vertex labeling

By multiple automorphisms (different traversal order for the same graph) on a single core
Candidate Generation Based On Core Detection - Issues (2)

Multiple cores between two \((k-1)\)-subgraphs
Candidate pruning: downward closure property

- Every \((k-1)\)-subgraph must be frequent.
- For all the \((k-1)\)-subgraphs of a given \(k\)-candidate, check if downward closure property holds

3-candidates: 

4-candidates:
frequent 1-subgraphs

frequent 2-subgraphs

3-candidates

frequent 3-subgraphs

4-candidates

frequent 4-subgraphs
Candidate Pruning: Canonical Labeling

**Idea**: Use the adjacency matrix to construct a hashable string representation of the graph
- Concatenate the columns (if undirected only of the upper right matrix)

Code($M_1$) = “aabyzx”

Code($M_2$) = “abaxyz”

Graph $G$:

Canonical-Code($G$) = $\min \{ \text{code}(M) \mid M \text{ is adj. Matrix} \}$
Canonical labeling

- **Intuitively**: Find a unique canonical form (relabeling) or automorphism (a mapping among nodes in the same graph) such that the canonical form for two isomorphic graphs is the same.

- The problem is as complex as graph isomorphism, but FSG suggests some heuristics to speed it up such as:
  - Vertex Invariants (e.g. degree)
  - Neighbor lists
  - Iterative partitioning
**Frequency counting**

**Idea:** For each pattern $f_1, \ldots, f_r$ take a list of graphs TID that contain the pattern
- When joining two patterns compute the intersection between the lists
- If the size of the intersection $< \text{min\_supp}$ remove the pattern

\[
G_1 = \{f_1, f_2, f_3\} \\
G_2 = \{f_1\} \\
G_3 = \{f_2\}
\]

\[
\text{TID}(f_1) = \{G_1, G_2\} \\
\text{TID}(f_2) = \{G_1, G_3\}
\]

Candidate
\[
c = \text{join}(f_1, f_2)
\]

\[
\text{TID}(c) = \text{subset}(\text{TID}(f_1) \cap \text{TID}(f_2))
\]

Perform subgraph isomorphism only on $G_1$
Simple operations?

- **Candidate generation**
  - To determine two candidates for joining, we need to check for graph isomorphism.
  - **Solution:** use Core detection

- **Candidate pruning**
  - To check downward closure property, we need graph isomorphism.
  - **Solution:** use Canonical labeling

- **Frequency counting**
  - Subgraph isomorphism for checking containment of a frequent subgraph.
  - **Solution:** use TID lists

Simpler? Yes with some graphs but in general still NP-complete
Mining approaches: agenda

- **Apriori-based approaches:**
  - FSG

- **Pattern-growth approaches:**
  - gSpan

- **Greedy approach:**
  - Subdue
gSpan

- One of the earliest and most used approaches for subgraph mining
- Makes use of the properties of (Depth First Search) DFS traversals to define canonical codes called DFS-codes
- Reduces the redundancy in the generation of the patterns

Yan, X. and Han, J., 2002. gSpan: Graph-based substructure pattern mining. *ICDM 2003*
Motivation: DFS exploration wrt. itemsets.

Itemset search space $\rightarrow$ prefix based

Prefix based exploration in graphs? DFS

Graph:

- abcde
- abcd, abce, abde, acde
- abc, abd, abe, acd, ace, ade
- bc, bd, be, cd, ce, de
- a, b, c, d, e
Why using a prefix tree?

- Canonical representation of itemset is obtained by a complete order over the items.
- Each possible itemset appear in the tree exactly once - no duplications or omissions.
  - The algorithm is complete and correct

Properties of Tree search space
  - for each k-label, its parent is the k-1 prefix of the given k-label
  - The relation among siblings is in ascending lexicographic order.
DFS Code representation

- Map each graph (2-Dim) to a sequential DFS Code (1-Dim)
- Lexicographically ordered codes
- Construct Tree Search Space based on the lexicographic order.

![Graph Diagram]

<table>
<thead>
<tr>
<th>Edge#</th>
<th>Code</th>
<th>DFS-edge: ((i, j, L_i, L_{i,j}, L_j))</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(0,1,X,a,Y)</td>
<td>i,j – vertices by discovery time</td>
</tr>
<tr>
<td>1</td>
<td>(1,2,Y,b,X)</td>
<td>(L_i, L_j) - vertex labels of (v_i, v_j)</td>
</tr>
<tr>
<td>2</td>
<td>(2,0,X,a,X)</td>
<td>(L_{i,j}) - edge label between (v_i, v_j)</td>
</tr>
<tr>
<td>3</td>
<td>(2,3,X,c,Z)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>(3,1,Z,b,Y)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>(1,4,Y,d,Z)</td>
<td></td>
</tr>
</tbody>
</table>

Vertex discovery time
DFS Code construction

- Given a graph $G$, for each Depth First Search over graph $G$, construct the corresponding DFS-Code.

\[(0,1,X,a,Y) \quad (1,2,Y,b,X) \quad (2,0,X,a,X) \quad (2,3,X,c,Z) \quad (3,1,Z,b,Y) \quad (1,4,Y,d,Z)\]
### Single graph, several DFS-Codes

<table>
<thead>
<tr>
<th></th>
<th>(a)</th>
<th>(b)</th>
<th>(c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(0, 1, X, a, Y)</td>
<td>(0, 1, Y, a, X)</td>
<td>(0, 1, X, a, X)</td>
</tr>
<tr>
<td>2</td>
<td>(1, 2, Y, b, X)</td>
<td>(1, 2, X, a, X)</td>
<td>(1, 2, X, a, Y)</td>
</tr>
<tr>
<td>3</td>
<td>(2, 0, X, a, X)</td>
<td>(2, 0, X, b, Y)</td>
<td>(2, 0, Y, b, X)</td>
</tr>
<tr>
<td>4</td>
<td>(2, 3, X, c, Z)</td>
<td>(2, 3, X, c, Z)</td>
<td>(2, 3, Y, b, Z)</td>
</tr>
<tr>
<td>5</td>
<td>(3, 1, Z, b, Y)</td>
<td>(3, 0, Z, b, Y)</td>
<td>(3, 0, Z, c, X)</td>
</tr>
<tr>
<td>6</td>
<td>(1, 4, Y, d, Z)</td>
<td>(0, 4, Y, d, Z)</td>
<td>(2, 4, Y, d, Z)</td>
</tr>
</tbody>
</table>

![Graph with nodes X, Y, Z, a, b, c, d and edges connecting them.](image)
Valid DFS-Code Edge ordering

Define a specific order on edges corresponding to the DFS traversal

- \( e_1 = (i_1, j_1), e_2 = (i_2, j_2) \)
- \( e_1 < e_2 \Rightarrow e_1 \text{ appears before } e_2 \text{ in the code} \)

Ordering rules

1. If it is a backward edge, \( i_1 = i_2 \) and \( j_1 < j_2 \) \( \Rightarrow e_1 < e_2 \)
2. If it is a forwards edge, \( i_1 < j_1 \) and \( j_1 = i_2 \) \( \Rightarrow e_1 < e_2 \)
3. If \( e_1 < e_2 \) and \( e_2 < e_3 \) \( \Rightarrow e_1 < e_3 \) (transitive property)

Why enforcing an edge ordering in the DFS code?

You want to ensure that the code (and so the DFS exploration) is produced in a certain order for later comparisons in a prefix style.
Multiple codes?

- Having multiple codes for the same graph is not good, since we don’t have an effective way to compare two graphs

- **Idea**: find (define) a total order between DFS-codes, so we can find the minimum (or the maximum)
  - If we compare the minimum DFS-codes of two graphs we are sure that if they are equal then the graphs are isomorphic
### Single graph - single Min DFS-code

<table>
<thead>
<tr>
<th></th>
<th>(a)</th>
<th>(b)</th>
<th>(c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(0, 1, X, a, Y)</td>
<td>(0, 1, Y, a, X)</td>
<td>(0, 1, X, a, X)</td>
</tr>
<tr>
<td>2</td>
<td>(1, 2, Y, b, X)</td>
<td>(1, 2, X, a, X)</td>
<td>(1, 2, X, a, Y)</td>
</tr>
<tr>
<td>3</td>
<td>(2, 0, X, a, X)</td>
<td>(2, 0, X, b, Y)</td>
<td>(2, 0, Y, b, X)</td>
</tr>
<tr>
<td>4</td>
<td>(2, 3, X, c, Z)</td>
<td>(2, 3, X, c, Z)</td>
<td>(2, 3, Y, b, Z)</td>
</tr>
<tr>
<td>5</td>
<td>(3, 1, Z, b, Y)</td>
<td>(3, 0, Z, b, Y)</td>
<td>(3, 0, Z, c, X)</td>
</tr>
<tr>
<td>6</td>
<td>(1, 4, Y, d, Z)</td>
<td>(0, 4, Y, d, Z)</td>
<td>(2, 4, Y, d, Z)</td>
</tr>
</tbody>
</table>
What is minimum?

- In order to define a minimum we have to define an order on the DFS codes.
- Assume you have an order on the edge/node labels.
- Given two DFS codes $\alpha, \beta$ for two graphs, how can we determine if $\alpha < \beta$?
  - $\alpha = (a_0, a_1, ..., a_m), \beta = (b_0, b_1, ..., b_n)$
  - Assume $n \geq m$
- $\alpha \leq \beta$ iff either of the following are true:
  - $\exists t, 0 \leq t \leq \min(n, m)$ such that $a_k = b_k$ for $k < t$ and $a_t < e b_t$
  - $a_k = b_k$ for $0 \leq k \leq m$
Defining $a_t \prec_e b_t$

- $a_t = (i_a, j_a, L_i_a, L_{i_a,j_a}, L_{j_a})$
- $b_t = (i_b, j_b, L_i_b, L_{i_b,j_b}, L_{j_b})$
- $a_t \prec_e b_t$ if
  1. Both are forward edges and
     - $i_b < i_a$ (edge starts from a later visited vertex, why?)
     - $i_b = i_a$ and the labels of $a$ are lexicographically less than labels of $b$, in order of tuple
  2. Both are backward edges ($i_a = i_b$ for the same reason)
     - $j_a < j_b$ (edge connected to an earlier discovered vertex)
     - $j_a = j_b$ and the edge label of $a$ is lexicographically less than the one of $b$

Why not the node labels?

3. $a_t$ is backward and $b_t$ is forward

If they are forward edges $j_a = j_b$ because since the previous edges are equal you have discovered a new node
The minimum DFS code $\text{min-DFS}(G)$ for a graph $G$ in DFG lexicographic order, is a canonical representation of graph $G$.

**Theorem**

Graphs $G_1$ and $G_2$ are isomorphic iff

$$\text{min-DFS}(G_1) = \text{min-DFS}(G_2)$$
DFS-Code Tree: parent-child relation

- If \( \min(G_1) = \{ a_0, a_1, \ldots, a_n \} \)
  and \( \min(G_2) = \{ a_0, a_1, \ldots, a_n, b \} \)
  
  - \( G_1 \) is parent of \( G_2 \)
  - \( G_2 \) is child of \( G_1 \)

- A valid DFS code requires that \( b \) (because \( b \) by definition is larger than any \( a_i \)) grows from a vertex on the rightmost path (inherited property from the DFS search).
  - Forward edge extensions to a DFS code must occur from a vertex on the rightmost path
  - Backward edge extensions must occur from the rightmost vertex.

If it is NOT in the rightmost path then it has been already discovered by the DFS.
Graph $G_1$:

Min($g$) = (0,1,X,a,Y) (1,2,Y,b,X) (2,0,X,a,X) (2,3,X,c,Z) (3,1,Z,b,Y) (1,4,Y,d,Z)

A child of graph $G_1$ must grow edge from rightmost path of $G_1$ (necessary condition)

Graph $G_2$:

Wrong

Forward EDGE

Backward EDGE

Only from the rightmost node!
Search space: DFS code Tree

- Organize DFS Code nodes as parent-child rules
  - Nodes are DFS codes except for the first level of the tree in which a vertex represents a (frequent) vertex label
  - Each level of the tree adds an edge to the DFS code
  - Sibling nodes organized in ascending DFS lexicographic order.
  - **InOrder traversal follows DFS lexicographic order!**
  - Backward edges are expanded first, why? Think about the rules for the DFS-codes ;)

GRAPH MINING WS 2017
Two pruning rules:
1. The code is not minimum
2. The support is < min_support
gSpan Algorithm

- Traverse DFS code tree for given label sets
  - Prune using support, minimality of codes
- **Input**: Graph database D, min_sup
- **Output**: frequent subgraphs set S
- **Procedure**:
  1. $S \leftarrow$ frequent one-edge subgraphs in D (using DFS code)
  2. Sort $S$ in lexicographic order
  3. $N \leftarrow S$ ($S$ will be modified)
  4. for each $n \in N$ do:
     - gSpan_Extend ($D, n, \text{min\_sup}, S$)
  5. Remove $n$ from all graphs in D (consider subgraphs not already enumerated)
- **Strategy**: grow minimal DFS codes that occur frequently in D
gSpan_Extend

- **Input**: Graph database \( D \), \( \text{min\_sup} \), DFS code \( n \)
- **Output**: frequent subgraph set \( S \)
- **Procedure**
  1. If \( n \) not minimal end
  2. Otherwise
     1. Add \( n \) to \( S \)
     2. for each single-edge rightmost expansion \( e \) of \( n \)
        1. If \( \sigma(e) \geq \text{min\_sup} \) then gSpan_Extend(\( D, e, \text{min\_sup}, S \))
Mining approaches: agenda

- Apriori-based approaches:
  - FSG

- Pattern-growth approaches:
  - gSpan

- Greedy approach (in brief):
  - Subdue
Subdue algorithm

- A greedy algorithm for finding some of the most prevalent subgraphs.
- This method is not complete, i.e. it may not obtain all frequent subgraphs, although it pays in fast execution.
- Based on the description length: compresses graphs using graph patterns:
  - Use the patterns that give the maximum compression
- Based on Beam Search - like BFS it progresses level by level. Unlike BFS, however, beam search moves downward only through the best W (beam width) nodes at each level. The other nodes are ignored.
Any questions?
References

- T. Washio A. Inokuchi and H. Motoda, An Apriori-Based Algorithm for Mining Frequent Substructures from Graph Data, Proceedings of the 4th PKDD'00, 2000, pages 13-23.
- M. Kuramochi and G. Karypis, An Efficient Algorithm for Discovering Frequent Subgraphs, Tech. report, Department of Computer Science/Army HPC Research Center, 2002.
- N. Vanetik, E. Gudes, and S. E. Shimony, Computing Frequent Graph Patterns from Semistructured Data, Proceedings of the 2002 IEEE ICDM'02Y.
- Xifeng and H. Jiawei, gspan: Graph-Based Substructure Pattern Mining, Tech. report, University of Illinois at Urbana-Champaign, 2002.