Data Mining and Analysis: Fundamental Concepts and Algorithms

dataminingbook.info

Mohammed J. Zaki[1] Wagner Meira Jr.[2]

[1]Department of Computer Science
Rensselaer Polytechnic Institute, Troy, NY, USA

[2]Department of Computer Science
Universidade Federal de Minas Gerais, Belo Horizonte, Brazil

Chapter 11: Graph Pattern Mining

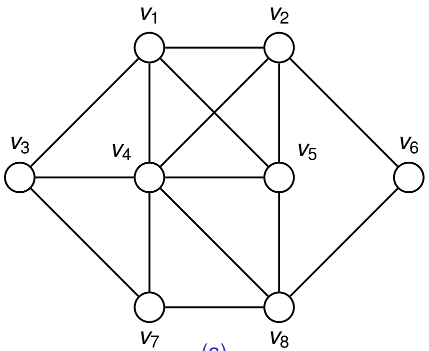
The goal of graph mining is to extract interesting subgraphs from a single large graph (e.g., a social network), or from a database of many graphs.

A graph is a pair $G = (V, E)$ where V is a set of vertices, and $E \subseteq V \times V$ is a set of edges.

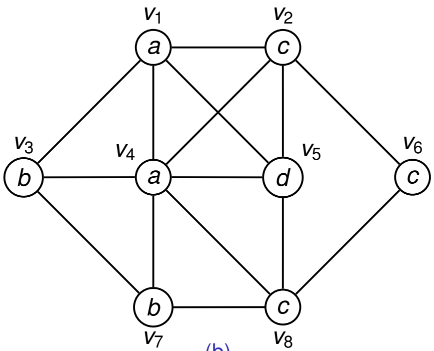
A *labeled graph* has labels associated with its vertices as well as edges. We use $L(u)$ to denote the label of the vertex u, and $L(u, v)$ to denote the label of the edge (u, v), with the set of vertex labels denoted as Σ_V and the set of edge labels as Σ_E. Given an edge $(u, v) \in G$, the tuple $\langle u, v, L(u), L(v), L(u, v)\rangle$ that augments the edge with the node and edge labels is called an *extended edge*.

A graph $G' = (V', E')$ is said to be a *subgraph* of G if $V' \subseteq V$ and $E' \subseteq E$. A *connected subgraph* is defined as a subgraph G' such that $V' \subseteq V$, $E' \subseteq E$, and for any two nodes $u, v \in V'$, there exists a *path* from u to v in G'.

Unlabeled and Labeled Graph

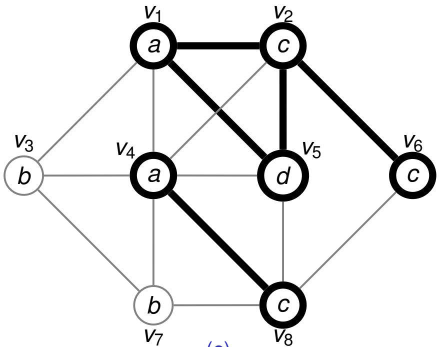


(a)

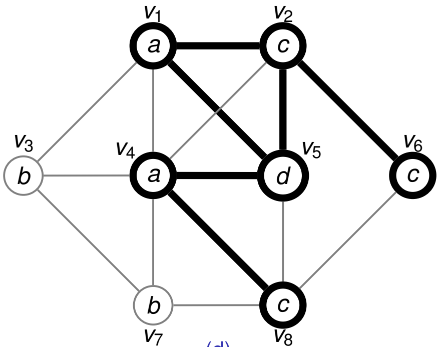


(b)

Subgraph and Connected Subgraph



(c)



(d)

Graph and Subgraph Isomorphism

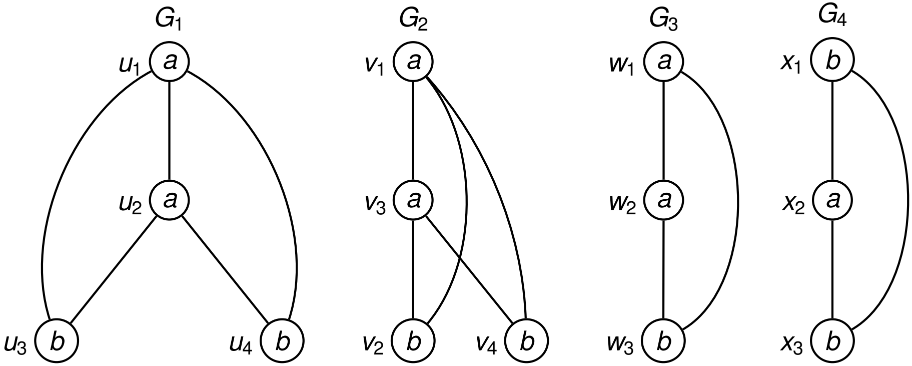
A graph $G' = (V', E')$ is said to be *isomorphic* to another graph $G = (V, E)$ if there exists a bijective function $\phi : V' \to V$, i.e., both injective (into) and surjective (onto), such that

- $(u, v) \in E' \iff (\phi(u), \phi(v)) \in E$
- $\forall u \in V',\ L(u) = L(\phi(u))$
- $\forall (u, v) \in E',\ L(u, v) = L(\phi(u), \phi(v))$

In other words, the *isomorphism* ϕ preserves the edge adjacencies as well as the vertex and edge labels. Put differently, the extended tuple $\langle u, v, L(u), L(v), L(u, v)\rangle \in G'$ if and only if $\langle \phi(u), \phi(v), L(\phi(u)), L(\phi(v)), L(\phi(u), \phi(v))\rangle \in G$.

If the function ϕ is only injective but not surjective, we say that the mapping ϕ is a *subgraph isomorphism* from G' to G. In this case, we say that G' is isomorphic to a subgraph of G, that is, G' is *subgraph isomorphic* to G, denoted $G' \subseteq G$; we also say that G *contains* G'.

Graph and Subgraph Isomorphism



Graph Isomorphism

G_1 and G_2 are isomorphic graphs. There are several possible isomorphisms between G_1 and G_2. An example of an isomorphism $\phi : V_2 \rightarrow V_1$ is

$$\phi(v_1) = u_1 \qquad \phi(v_2) = u_3 \qquad \phi(v_3) = u_2 \qquad \phi(v_4) = u_4$$

The inverse mapping ϕ^{-1} specifies the isomorphism from G_1 to G_2. For example, $\phi^{-1}(u_1) = v_1$, $\phi^{-1}(u_2) = v_3$, and so on.

The set of all possible isomorphisms from G_2 to G_1 are as follows:

	v_1	v_2	v_3	v_4
ϕ_1	u_1	u_3	u_2	u_4
ϕ_2	u_1	u_4	u_2	u_3
ϕ_3	u_2	u_3	u_1	u_4
ϕ_4	u_2	u_4	u_1	u_3

Subgraph Isomorphism

The graph G_3 is subgraph isomorphic to both G_1 and G_2. The set of all possible subgraph isomorphisms from G_3 to G_1 are as follows:

	w_1	w_2	w_3
ϕ_1	u_1	u_2	u_3
ϕ_2	u_1	u_2	u_4
ϕ_3	u_2	u_1	u_3
ϕ_4	u_2	u_1	u_4

Mining Frequent Subgraph

Given a database of graphs, $\mathbf{D} = \{G_1, G_2, \ldots, G_n\}$, and given some graph G, the support of G in $\mathbf{D}$ is defined as follows:

$$sup(G) = \Big|\big\{G_i \in \mathbf{D} \mid G \subseteq G_i\big\}\Big|$$

The support is simply the number of graphs in the database that contain G. Given a *minsup* threshold, the goal of graph mining is to mine all frequent connected subgraphs with $sup(G) \geq minsup$.

If we consider subgraphs with m vertices, then there are $\binom{m}{2} = O(m^2)$ possible edges. The number of possible subgraphs with m nodes is then $O(2^{m^2})$ because we may decide either to include or exclude each of the edges. Many of these subgraphs will not be connected, but $O(2^{m^2})$ is a convenient upper bound. When we add labels to the vertices and edges, the number of labeled graphs will be even more.

Graph Pattern Mining

There are two main challenges in frequent subgraph mining.

The first is to systematically generate nonredundant candidate subgraphs. We use *edge-growth* as the basic mechanism for extending the candidates.

The second challenge is to count the support of a graph in the database. This involves subgraph isomorphism checking, as we have to find the set of graphs that contain a given candidate.

Candidate Generation

An effective strategy to enumerate subgraph patterns is *rightmost path extension*.

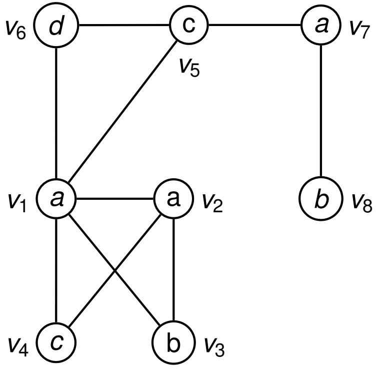
Given a graph G, we perform a depth-first search (DFS) over its vertices, and create a DFS spanning tree, that is, one that covers or spans all the vertices.

Edges that are included in the DFS tree are called *forward* edges, and all other edges are called *backward* edges. Backward edges create cycles in the graph.

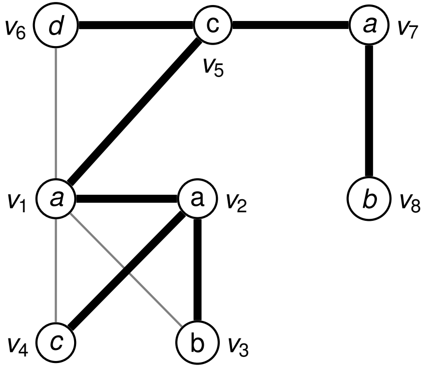
Once we have a DFS tree, define the *rightmost* path as the path from the root to the rightmost leaf, that is, to the leaf with the highest index in the DFS order.

Depth-first Spanning Tree

Starting at v_1, each DFS step chooses the vertex with smallest index



(e)



(f)

Rightmost Path Extensions

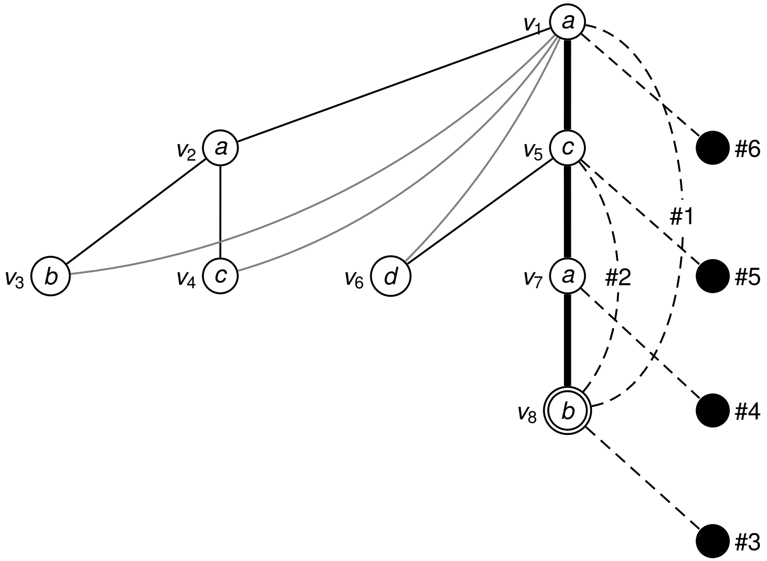
For generating new candidates from a given graph G, we extend it by adding a new edge to vertices only on the rightmost path. We can either extend G by adding backward edges from the *rightmost vertex* to some other vertex on the rightmost path (disallowing self-loops or multi-edges), or we can extend G by adding forward edges from any of the vertices on the rightmost path. A backward extension does not add a new vertex, whereas a forward extension adds a new vertex.

For systematic candidate generation we impose a total order on the extensions, as follows: First, we try all backward extensions from the rightmost vertex, and then we try forward extensions from vertices on the rightmost path.

Among the backward edge extensions, if u_r is the rightmost vertex, the extension (u_r, v_i) is tried before (u_r, v_j) if $i < j$. In other words, backward extensions closer to the root are considered before those farther away from the root along the rightmost path.

Among the forward edge extensions, if v_x is the new vertex to be added, the extension (v_i, v_x) is tried before (v_j, v_x) if $i > j$. In other words, the vertices farther from the root (those at greater depth) are extended before those closer to the root. Also note that the new vertex will be numbered $x = r + 1$, as it will become the new rightmost vertex after the extension.

Rightmost Path Extensions



DFS Code

For systematic enumeration we rank the set of isomorphic graphs and pick one member as the *canonical* representative.

Let G be a graph and let T_G be a DFS spanning tree for G. The DFS tree T_G defines an ordering of both the nodes and edges in G. The DFS node ordering is obtained by numbering the nodes consecutively in the order they are visited in the DFS walk.

Assume that for a pattern graph G the nodes are numbered according to their position in the DFS ordering, so that $i < j$ implies that v_i comes before v_j in the DFS walk.

The DFS edge ordering is obtained by following the edges between consecutive nodes in DFS order, with the condition that all the backward edges incident with vertex v_i are listed before any of the forward edges incident with it.

The *DFS code* for a graph G, for a given DFS tree T_G, denoted DFScode(G), is defined as the sequence of extended edge tuples of the form $\langle v_i, v_j, L(v_i), L(v_j), L(v_i, v_j)\rangle$ listed in the DFS edge order.

Canonical DFS Code

A subgraph is *canonical* if it has the smallest DFS code among all possible isomorphic graphs.

Let t_1 and t_2 be any two DFS code tuples:

$$t_1 = \langle v_i, v_j, L(v_i), L(v_j), L(v_i, v_j) \rangle$$
$$t_2 = \langle v_x, v_y, L(v_x), L(v_y), L(v_x, v_y) \rangle$$

We say that t_1 is smaller than t_2, written $t_1 < t_2$, iff

i) $(v_i, v_j) <_e (v_x, v_y)$, or

ii) $(v_i, v_j) = (v_x, v_y)$ and $\langle L(v_i), L(v_j), L(v_i, v_j) \rangle <_l \langle L(v_x), L(v_y), L(v_x, v_y) \rangle$

where $<_e$ is an ordering on the edges and $<_l$ is an ordering on the vertex and edge labels.

The *label order* $<_l$ is the standard lexicographic order on the vertex and edge labels.

The *edge order* $<_e$ is derived from the rules for rightmost path extension, namely that all of a node's backward extensions must be considered before any forward edge from that node, and deep DFS trees are preferred over bushy DFS trees.

Canonical DFS Code: Edge Ordering

Let $e_{ij} = (v_i, v_j)$ and $e_{xy} = (v_x, v_y)$ be any two edges. We say that $e_{ij} <_e e_{xy}$ iff

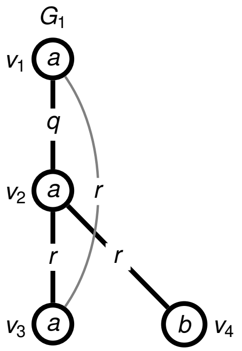
- If e_{ij} and e_{xy} are both forward edges, then (a) $j < y$, or (b) $j = y$ and $i > x$.
- If e_{ij} and e_{xy} are both backward edges, then (a) $i < x$, or (b) $i = x$ and $j < y$.
- If e_{ij} is a forward and e_{xy} is a backward edge, then $j \le x$.
- If e_{ij} is a backward and e_{xy} is a forward

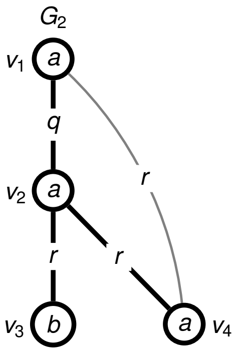
The *canonical DFS code* for a graph G is defined as follows:

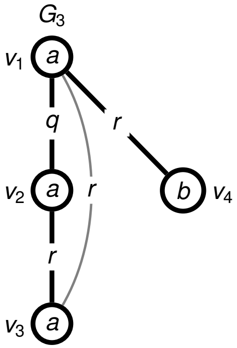
$$\mathcal{C} = \min_{G'} \Big\{ \text{DFScode}(G') \mid G' \text{ is isomorphic to } G \Big\}$$

Canonical DFS Code

G_1 has the canonical or minimal DFS code







$$\boxed{\begin{aligned} t_{11} &= \langle v_1, v_2, a, a, q\rangle \\ t_{12} &= \langle v_2, v_3, a, a, r\rangle \\ t_{13} &= \langle v_3, v_1, a, a, r\rangle \\ t_{14} &= \langle v_2, v_4, a, b, r\rangle \end{aligned}}$$

DFScode(G_1)

$$\boxed{\begin{aligned} t_{21} &= \langle v_1, v_2, a, a, q\rangle \\ t_{22} &= \langle v_2, v_3, a, b, r\rangle \\ t_{23} &= \langle v_2, v_4, a, a, r\rangle \\ t_{24} &= \langle v_4, v_1, a, a, r\rangle \end{aligned}}$$

DFScode(G_2)

$$\boxed{\begin{aligned} t_{31} &= \langle v_1, v_2, a, a, q\rangle \\ t_{32} &= \langle v_2, v_3, a, a, r\rangle \\ t_{33} &= \langle v_3, v_1, a, a, r\rangle \\ t_{34} &= \langle v_1, v_4, a, b, r\rangle \end{aligned}}$$

DFScode(G_3)

gSpan Graph Mining Algorithm

gSpan enumerates patterns in a depth-first manner, starting with the empty code. Given a canonical and frequent code C, gSpan first determines the set of possible edge extensions along the rightmost path.

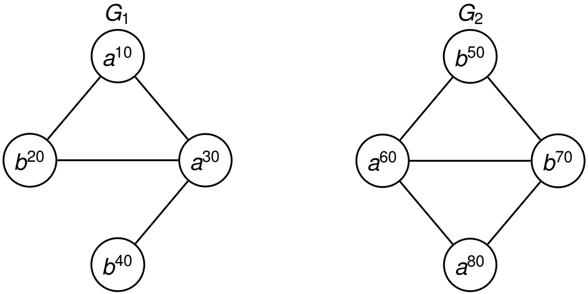
The function RIGHTMOSTPATH-EXTENSIONS returns the set of edge extensions along with their support values, $\mathcal{E}$. Each extended edge t in $\mathcal{E}$ leads to a new candidate DFS code $C' = C \cup \{t\}$, with support $sup(C) = sup(t)$.

For each new candidate code, gSpan checks whether it is frequent and canonical, and if so gSpan recursively extends C' The algorithm stops when there are no more frequent and canonical extensions possible.

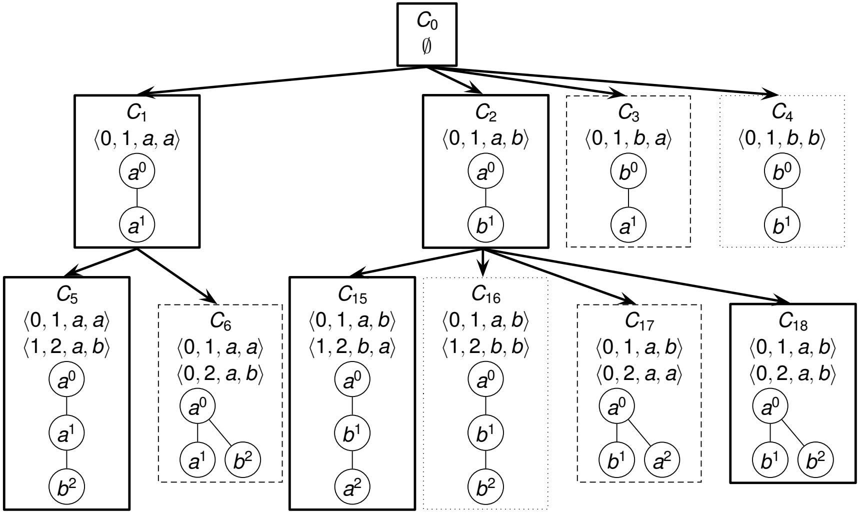
Algorithm gSpan

```
// Initial Call: $C \leftarrow \emptyset$
gSpan ($C$, D, $minsup$):
1 $\mathcal{E} \leftarrow$ RightMostPath-Extensions($C$, D) // extensions and
      supports
2 foreach $(t, sup(t)) \in \mathcal{E}$ do
3     $C' \leftarrow C \cup t$ // extend the code with extended edge
          tuple $t$
4     $sup(C') \leftarrow sup(t)$ // record the support of new
          extension
      // recursively call gSpan if code is frequent and
          canonical
5     if $sup(C') \geq minsup$ and IsCanonical ($C'$) then
6         gSpan ($C'$, D, $minsup$)
```

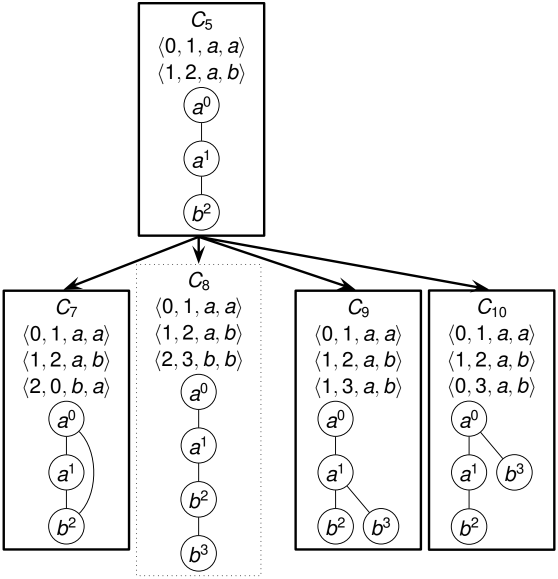
Example Graph Database: gSpan



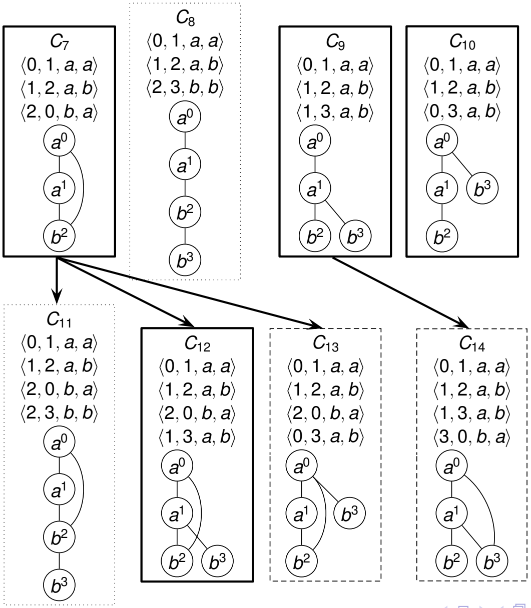
Frequent Graph Mining: gSpan



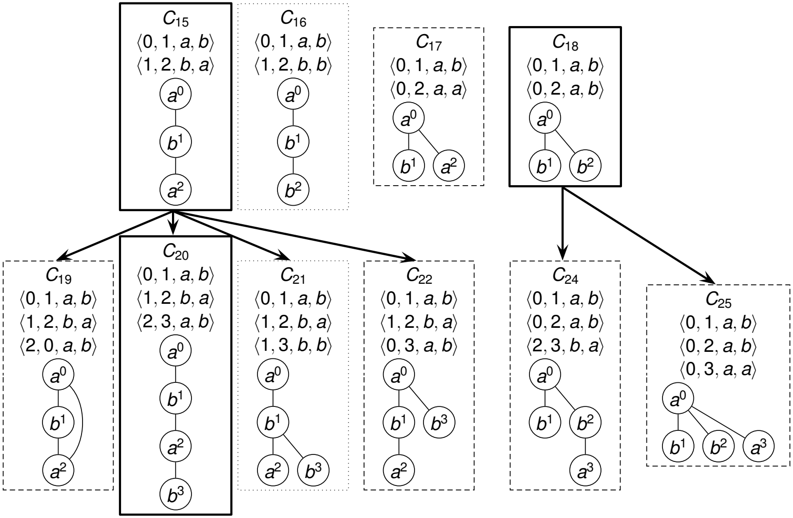
Frequent Graph Mining: gSpan



Frequent Graph Mining: gSpan

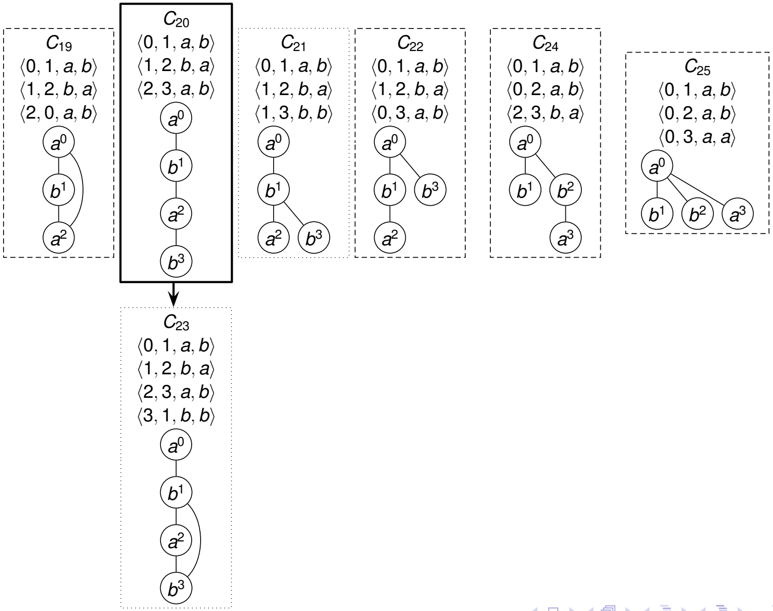


Frequent Graph Mining: gSpan



C_{15} $\langle 0,1,a,b\rangle$ $\langle 1,2,b,a\rangle$
C_{16} $\langle 0,1,a,b\rangle$ $\langle 1,2,b,b\rangle$
C_{17} $\langle 0,1,a,b\rangle$ $\langle 0,2,a,a\rangle$
C_{18} $\langle 0,1,a,b\rangle$ $\langle 0,2,a,b\rangle$
C_{19} $\langle 0,1,a,b\rangle$ $\langle 1,2,b,a\rangle$ $\langle 2,0,a,b\rangle$
C_{20} $\langle 0,1,a,b\rangle$ $\langle 1,2,b,a\rangle$ $\langle 2,3,a,b\rangle$
C_{21} $\langle 0,1,a,b\rangle$ $\langle 1,2,b,a\rangle$ $\langle 1,3,b,b\rangle$
C_{22} $\langle 0,1,a,b\rangle$ $\langle 1,2,b,a\rangle$ $\langle 0,3,a,b\rangle$
C_{24} $\langle 0,1,a,b\rangle$ $\langle 0,2,a,b\rangle$ $\langle 2,3,b,a\rangle$
C_{25} $\langle 0,1,a,b\rangle$ $\langle 0,2,a,b\rangle$ $\langle 0,3,a,a\rangle$

Frequent Graph Mining: gSpan



Extension and Support Computation

The support computation task is to find the number of graphs in the database **D** that contain a candidate subgraph, which is very expensive because it involves subgraph isomorphism checks. gSpan combines the tasks of enumerating candidate extensions and support computation.

Assume that $\mathbf{D} = \{G_1, G_2, \dots, G_n\}$ comprises n graphs. Let $C = \{t_1, t_2, \dots, t_k\}$ denote a frequent canonical DFS code comprising k edges, and let $G(C)$ denote the graph corresponding to code C. The task is to compute the set of possible rightmost path extensions from C, along with their support values.

Given code C, gSpan first records the nodes on the rightmost path (R), and the rightmost child (u_r). Next, gSpan considers each graph $G_i \in \mathbf{D}$. If $C = \emptyset$, then each distinct label tuple of the form $\langle L(x), L(y), L(x,y) \rangle$ for adjacent nodes x and y in G_i contributes a forward extension $\langle 0, 1, L(x), L(y), L(x,y) \rangle$ On the other hand, if C is not empty, then gSpan enumerates all possible subgraph isomorphisms Φ_i between the code C and graph G_i. Given subgraph isomorphism $\phi \in \Phi_i$, gSpan finds all possible forward and backward edge extensions, and stores them in the extension set $\mathcal{E}$.

Forward and Backward Extensions

Backward extensions are allowed only from the rightmost child u_r in C to some other node on the rightmost path R.

The method considers each neighbor x of $\phi(u_r)$ in G_i and checks whether it is a mapping for some vertex $v = \phi^{-1}(x)$ along the rightmost path R in C. If the edge (u_r, v) does not already exist in C, it is a new extension, and the extended tuple $b = \langle u_r, v, L(u_r), L(v), L(u_r, v) \rangle$ is added to the set of extensions $\mathcal{E}$, along with the graph id i that contributed to that extension.

Forward extensions are allowed only from nodes on the rightmost path R to new nodes. For each node u in R, the algorithm finds a neighbor x in G_i that is not in a mapping from some node in C. For each such node x, the forward extension $f = \langle u, u_r + 1, L(\phi(u)), L(x), L(\phi(u), x) \rangle$ is added to $\mathcal{E}$, along with the graph id i. Because a forward extension adds a new vertex to the graph $G(C)$, the id of the new node in C must be $u_r + 1$, that is, one more than the highest numbered node in C, which by definition is the rightmost child u_r.

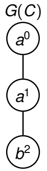
Algorithm RightMostPath-Extensions

```
RightMostPath-Extensions (C, D):
1  $R \leftarrow$ nodes on the rightmost path in $C$
2  $u_r \leftarrow$ rightmost child in $C$ // dfs number
3  $\mathcal{E} \leftarrow \emptyset$ // set of extensions from $C$
4  foreach $G_i \in \mathbf{D}, i = 1, \ldots, n$ do
5      if $C = \emptyset$ then
6          foreach distinct $\langle L(x), L(y), L(x,y) \rangle \in G_i$ do
7              $f = \langle 0, 1, L(x), L(y), L(x,y) \rangle$
8              Add tuple $f$ to $\mathcal{E}$ along with graph id $i$
9      else
10         $\Phi_i = \text{SubgraphIsomorphisms}(C, G_i)$
11         foreach isomorphism $\phi \in \Phi_i$ do
12             foreach $x \in N_{G_i}(\phi(u_r))$ such that $\exists v \leftarrow \phi^{-1}(x)$ do
13                 if $v \in R$ and $(u_r, v) \notin G(C)$ then
14                     $b = \langle u_r, v, L(u_r), L(v), L(u_r, v) \rangle$
15                     Add tuple $b$ to $\mathcal{E}$ along with graph id $i$
16             foreach $u \in R$ do
17                 foreach $x \in N_{G_i}(\phi(u))$ and $\nexists \phi^{-1}(x)$ do
18                     $f = \langle u, u_r + 1, L(\phi(u)), L(x), L(\phi(u), x) \rangle$
19                     Add tuple $f$ to $\mathcal{E}$ along with graph id $i$
20 foreach distinct extension $s \in \mathcal{E}$ do
21     $sup(s)$ = number of distinct graph ids that support tuple $s$
22 return set of pairs $\langle s, sup(s) \rangle$ for extensions $s \in \mathcal{E}$, in tuple sorted order
```

Righmost Path Extensions

C

$t_1 : \langle 0, 1, a, a\rangle$
$t_2 : \langle 1, 2, a, b\rangle$



(a) Code C and graph $G(C)$

Φ	ϕ	0	1	2
Φ_1	ϕ_1	10	30	20
	ϕ_2	10	30	40
	ϕ_3	30	10	20
Φ_2	ϕ_4	60	80	70
	ϕ_5	80	60	50
	ϕ_6	80	60	70

(b) Subgraph isomorphisms

Id	ϕ	Extensions
G_1	ϕ_1	$\{\langle 2,0,b,a\rangle, \langle 1,3,a,b\rangle\}$
	ϕ_2	$\{\langle 1,3,a,b\rangle, \langle 0,3,a,b\rangle\}$
	ϕ_3	$\{\langle 2,0,b,a\rangle, \langle 0,3,a,b\rangle\}$
G_2	ϕ_4	$\{\langle 2,0,b,a\rangle, \langle 2,3,b,b\rangle, \langle 0,3,a,b\rangle\}$
	ϕ_5	$\{\langle 2,3,b,b\rangle, \langle 1,3,a,b\rangle\}$
	ϕ_6	$\{\langle 2,0,b,a\rangle, \langle 2,3,b,b\rangle, \langle 1,3,a,b\rangle\}$

(c) Edge extensions

Extension	Support
$\langle 2,0,b,a\rangle$	2
$\langle 2,3,b,b\rangle$	1
$\langle 1,3,a,b\rangle$	2
$\langle 0,3,a,b\rangle$	2

(d) Extensions (sorted) and supports

Algorithm SubgraphIsomorphisms

```
   SubgraphIsomorphisms ($C = \{t_1, t_2, \dots, t_k\}$, $G$):
1  $\Phi \leftarrow \{\phi(0) \rightarrow x \mid x \in G$ and $L(x) = L(0)\}$
2  foreach $t_i \in C, i = 1, \dots, k$ do
3      $\langle u, v, L(u), L(v), L(u,v) \rangle \leftarrow t_i$ // expand extended edge $t_i$
4      $\Phi' \leftarrow \emptyset$ // partial isomorphisms including $t_i$
5      foreach partial isomorphism $\phi \in \Phi$ do
6          if $v > u$ then
               // forward edge
7              foreach $x \in N_G(\phi(u))$ do
8                  if $\nexists \phi^{-1}(x)$ and $L(x) = L(v)$ and $L(\phi(u), x) = L(u,v)$ then
9                      $\phi' \leftarrow \phi \cup \{\phi(v) \rightarrow x\}$
10                     Add $\phi'$ to $\Phi'$
11         else
               // backward edge
12             if $\phi(v) \in N_{G_j}(\phi(u))$ then Add $\phi$ to $\Phi'$ // valid isomorphism
13     $\Phi \leftarrow \Phi'$ // update partial isomorphisms
14 return $\Phi$
```

Subgraph Isomorphisms

To enumerate all the possible isomorphisms from C to each graph $G_i \in \mathbf{D}$ the function SUBGRAPHISOMORPHISMS, accepts a code C and a graph G, and returns the set of all isomorphisms between C and G.

The set of isomorphisms Φ is initialized by mapping vertex 0 in C to each vertex x in G that shares the same label as 0, that is, if $L(x) = L(0)$.

The method considers each tuple t_i in C and extends the current set of partial isomorphisms. Let $t_i = \langle u, v, L(u), L(v), L(u, v)\rangle$. We have to check if each isomorphism $\phi \in \Phi$ can be extended in G using the information from t_i

If t_i is a forward edge, then we seek a neighbor x of $\phi(u)$ in G such that x has not already been mapped to some vertex in C, that is, $\phi^{-1}(x)$ should not exist, and the node and edge labels should match, that is, $L(x) = L(v)$, and $L(\phi(u), x) = L(u, v)$. If so, ϕ can be extended with the mapping $\phi(v) \rightarrow x$. The new extended isomorphism, denoted ϕ', is added to the initially empty set of isomorphisms Φ'.

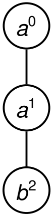
If t_i is a backward edge, we have to check if $\phi(v)$ is a neighbor of $\phi(u)$ in G. If so, we add the current isomorphism ϕ to Φ'.

Subgraph Isomorphisms

C

$t_1 : \langle 0, 1, a, a \rangle$
$t_2 : \langle 1, 2, a, b \rangle$

$G(C)$



Initial Φ

id	ϕ	0
G_1	ϕ_1	10
	ϕ_2	30
G_2	ϕ_3	60
	ϕ_4	80

Add t_1

id	ϕ	0, 1
G_1	ϕ_1	10, 30
	ϕ_2	30, 10
G_2	ϕ_3	60, 80
	ϕ_4	80, 60

Add t_2

id	ϕ	0, 1, 2
G_1	ϕ_1'	10, 30, 20
	ϕ_1''	10, 30, 40
	ϕ_2	30, 10, 20
G_2	ϕ_3	60, 80, 70
	ϕ_4'	80, 60, 50
	ϕ_4''	80, 60, 70

Canonicality Checking

Given a DFS code $C = \{t_1, t_2, \ldots, t_k\}$ comprising k extended edge tuples and the corresponding graph $G(C)$, the task is to check whether the code C is canonical.

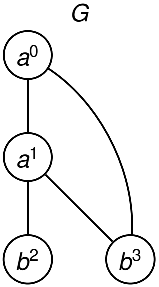
This can be accomplished by trying to reconstruct the canonical code C^* for $G(C)$ in an iterative manner starting from the empty code and selecting the least rightmost path extension at each step, where the least edge extension is based on the extended tuple comparison operator in

If at any step the current (partial) canonical DFS code C^* is smaller than C, then we know that C cannot be canonical and can thus be pruned. On the other hand, if no smaller code is found after k extensions then C must be canonical.

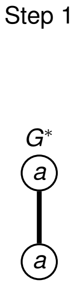
Algorithm IsCanonical: Canonicality Checking

```
IsCanonical ($C$):
1 $\mathbf{D}_C \leftarrow \{G(C)\}$ // graph corresponding to code $C$
2 $C^* \leftarrow \emptyset$ // initialize canonical DFScode
3 for $i = 1 \cdots k$ do
4     $\mathcal{E} = $ RightMostPath-Extensions$(C^*, \mathbf{D}_C)$ // extensions of $C^*$
5     $(s_i, sup(s_i)) \leftarrow \min\{\mathcal{E}\}$ // least rightmost edge extension of $C^*$
6     if $s_i < t_i$ then
7         return false // $C^*$ is smaller, thus $C$ is not canonical
8     $C^* \leftarrow C^* \cup s_i$
9 return true // no smaller code exists; $C$ is canonical
```

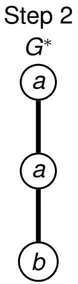
Canonicality Checking



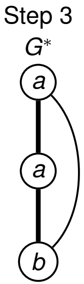
C
$t_1 = \langle 0, 1, a, a \rangle$
$t_2 = \langle 1, 2, a, b \rangle$
$t_3 = \langle 1, 3, a, b \rangle$
$t_4 = \langle 3, 0, b, a \rangle$



C^*
$s_1 = \langle 0, 1, a, a \rangle$



C^*
$s_1 = \langle 0, 1, a, a \rangle$
$s_2 = \langle 1, 2, a, b \rangle$



C^*
$s_1 = \langle 0, 1, a, a \rangle$
$s_2 = \langle 1, 2, a, b \rangle$
$s_3 = \langle 2, 0, b, a \rangle$