Chapter 11

Graph Pattern Mining

Graph data is becoming increasingly more ubiquitous in today's networked world. Examples include, social networks such as MySpace and Facebook as well as cell phone networks and blogs. The network routing across the Internet is another example of graph data, as is the hyperlinked structure of the world wide web (WWW). Bioinformatics, especially systems biology, deals with understanding interaction networks between various types of biomolecules, such as protein-protein interactions, metabolic networks, gene networks, and so on. Another example comes from semi-structured data, say, in the form of XML documents. XML has become one of the dominant formats for representation of documents on the web, and XML documents typically have a hierarchical structure, in the form of a tree (which is a rooted, connected, acyclic graph).

The goal of graph mining is to extract interesting subgraphs from a single large graph (e.g., the WWW), or from a database of many graphs. In different applications we may be interested in different kinds of subgraph patterns, such as subtrees, complete graphs or cliques, bipartite cliques, dense subgraphs, and so on. These may represent, for example, communities in a social network, hub and authority pages on the WWW, cluster of proteins involved in similar biochemical functions, and so on. Often, we may want to mine all the frequent subgraphs that appear in a database, without specifying a particular type of subgraph of interest.

11.1 Isomorphism and Support

A graph is a pair $G = (V, E)$ where V is a set of vertices, and E is a set of edges, where an edge is an unordered pair of distinct vertices, written as $(x, y) \in E$, where $x, y \in V$. If (x, y) is an edge, we say that x and y are *adjacent* or that y is a *neighbor* of x. The set of all neighbors of node x in G is given as $N_G(x) = \{y \in V \mid (x, y) \in E\}$. We say that a vertex is *incident* with an edge if it is one of the two vertices of that edge. A *labeled graph* has labels associated with its vertices as well as edges. We use $L(x)$ to denote the label of the vertex x and $L(x, y)$ to denote the label of

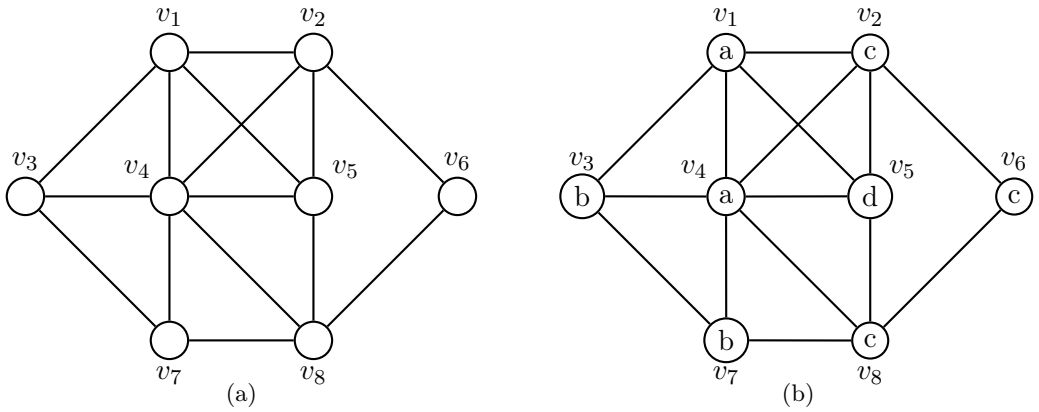


Figure 11.1: An unlabeled (a) and labeled (b) graph with 8 vertices

the edge (x, y), with the set of vertex labels denoted as Σ_V, and the set of edge labels as Σ_E. Given an edge $(x, y) \in G$, the corresponding *extended edge*, denoted $\langle x, y, L(x), L(y), L(x, y)\rangle$, augments the edge with the node and egde labels.

Example 11.1: Figure 11.1a shows an example of an unlabeled graph, whereas Figure 11.1b shows the same graph, with labels on the vertices, taken from the vertex label set $\Sigma_V = \{a, b, c, d\}$. In this example, edges are all assumed to have the same label, and are therefore not shown. Considering Figure 11.1b, the label of v_4 is $L(v_4) = a$, and its neighbors are given as $N(v_4) = \{v_1, v_2, v_3, v_5, v_7, v_8\}$. The edge (v_4, v_1) leads to the extended edge $\langle v_4, v_1, a, a\rangle$, where we omit the edge label $L(v_4, v_1)$.

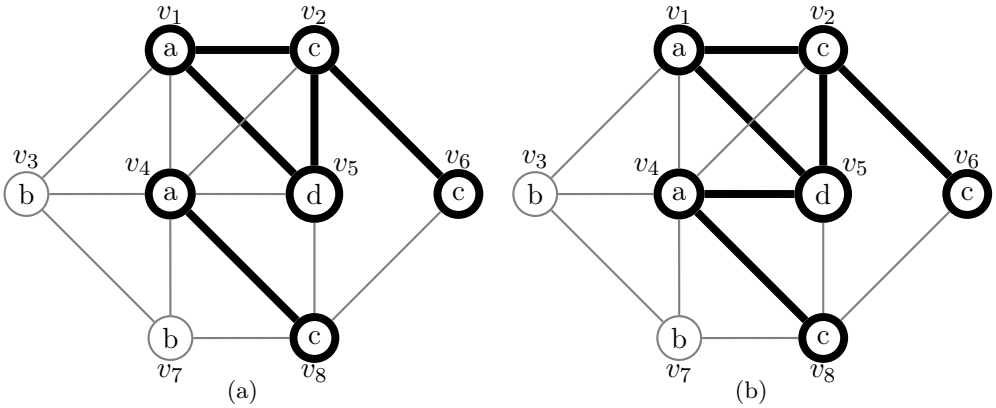


Figure 11.2: A subgraph (a) and connected subgraph (b)

Subgraphs: A graph $G' = (V', E')$ is said to be a *subgraph* of G if $V' \subseteq V$ and $E' \subseteq E$. Note that this definition allows for disconnected subgraphs. However, typically data mining applications call for *connected subgraphs*, defined as a subgraph G' such that $V' \subseteq V$, $E' \subseteq E$, and for any two nodes $x, y \in V'$, there exists a *path* from x to y in G'.

Example 11.2: The graph defined by bold edges in Figure 11.2a is a subgraph of the larger graph. However, it is a disconnected subgraph. Figure 11.2b shows an example of a connected subgraph.

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

1. $(x_1, x_2) \in E' \iff (\phi(x_1), \phi(x_2)) \in E$
2. $\forall x \in V',\ L(x) = L(\phi(x))$
3. $\forall (x_1, x_2) \in E',\ L(x_1, x_2) = L(\phi(x_1), \phi(x_2))$

In other words the *isomorphism* ϕ preserves the edge adjacencies, as well as the vertex and edge labels.

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

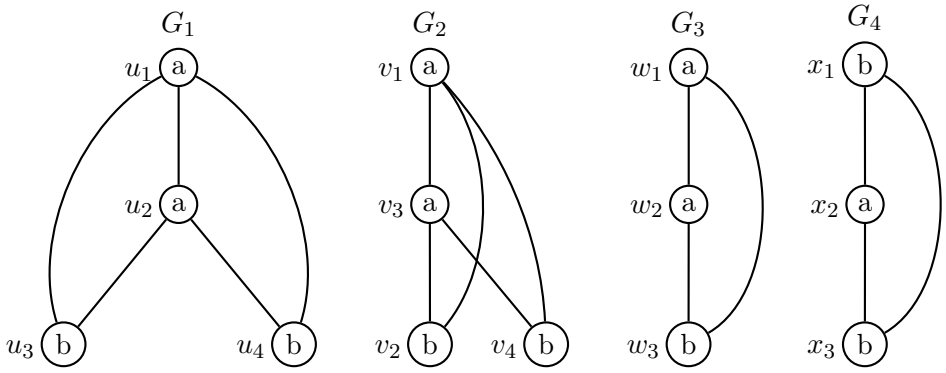


Figure 11.3: Graph and Subgraph Isomorphism

Example 11.3: In Figure 11.3, $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are isomorphic graphs. There are several possible isomorphisms between G_1 and G_2. An example of an isomorphism is $\phi : V_2 \rightarrow V_1$, defined as

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

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

The graph G_3 is subgraph isomorphic to both G_1 and G_2. For instance, 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

The graph G_4 is not subgraph isomorphic to either G_1 or G_2, and it is also not isomorphic to G_3, since the extended edge $\langle x_1, x_3, b, b\rangle$ has no possible mappings in G_1, G_2 or G_3.

Subgraph Support: Given a database of graphs, $\mathcal{D} = \{G_1, G_2, \cdots, G_n\}$, and given some graph G, we define the support of G in $\mathcal{D}$ as follows

$$sup(G) = \big|\{G_i \in \mathcal{D} \mid G \subseteq G_i\}\big| \tag{11.1}$$

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

To mine all the frequent subgraphs, one has to search over the space of all possible graph patterns, which is exponential in size. For example, if we consider graphs with m vertices, then there are $\binom{m}{2} = O(m^2)$ possible edges. The number of possible subgraphs with m nodes is then $2^{O(m^2)}$, since we may either decide to include or exclude each of the $O(m^2)$ edges. Many of these will not be connected, but this

provides an easy upper bound. When we add labels to the vertices and edges, the number of labeled graphs will be even more. Assume that $|\Sigma_V| = |\Sigma_E| = s$, then there are s^m possible ways to label the vertices and there are s^{m^2} ways to label the edges. Thus the number of possible labeled subgraphs with m vertices is $2^{O(m^2)} s^m s^{m^2} = O((2 \cdot s)^{O(m^2)})$. This is the worst case bound, since many of these subgraphs will be isomorphic to each other, so the number of distinct subgraphs will be much less. Nevertheless, the search space is still enormous, since we typically have to search for all subgraphs ranging from a single vertex to some maximum number of vertices, given by the largest frequent subgraph.

There are two main challenges in frequent subgraph mining. The first is to systematically generate candidate subgraphs. We use *edge-growth* as the basic mechanism for extending the candidates. The mining process proceeds in a breadth-first (level-wise) or a depth-first manner, starting with an empty subgraph (i.e., with no edge), and adding one new edge at a time. Such an edge may either connect two existing vertices in the graph or it may introduce a new vertex at one end of the edge. The key is to perform non-redundant subgraph enumeration, such that we do not generate the same graph candidate more than once. This means that we have to do graph isomorphism checking to make sure that duplicate graphs are removed. The second challenge is to count the support of a graph in the database. This involves subgraph isomorphism checking, since we have to find out the set of graphs that contain a given candidate.

11.2 Candidate Generation

11.2.1 Rightmost Path Extension

An effective strategy to enumerate subgraph patterns is the so-called *rightmost path extension*. Given a graph G, we perform a depth first search (DFS) over its vertices, and create a DFS spanning tree, i.e., 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 (or to the leaf with the highest index in the DFS order).

Example 11.4: Consider the graph shown in Figure 11.4a. One of the possible DFS spanning trees is shown in Figure 11.4b (illustrated via bold edges), obtained by starting at v_1 and then choosing the vertex with the smallest index at each step. Figure 11.5 shows the same graph (ignoring the dashed edges), rearranged to emphasize the DFS tree structure. For instance, the edges (v_1, v_2) and (v_2, v_3) are examples of forward edges, whereas (v_3, v_1), (v_4, v_1), and (v_6, v_1) are all backward edges. The bold edges (v_1, v_5), (v_5, v_7) and (v_7, v_8) comprise the rightmost path.

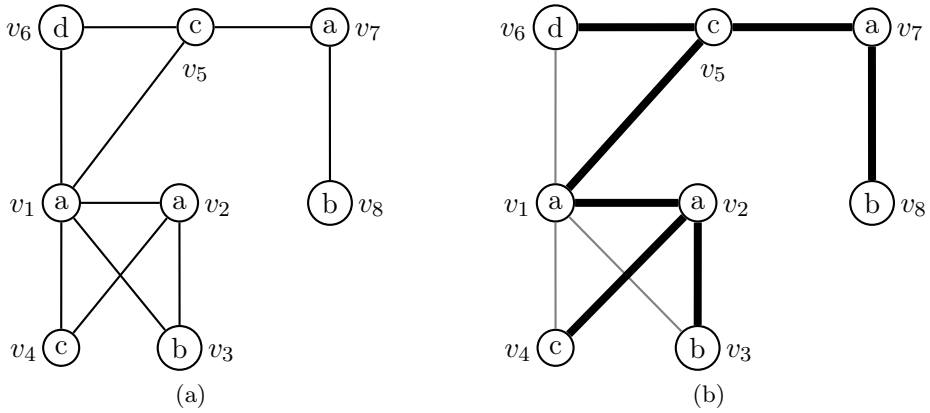


Figure 11.4: A graph (a) and a possible depth-first spanning tree (b)

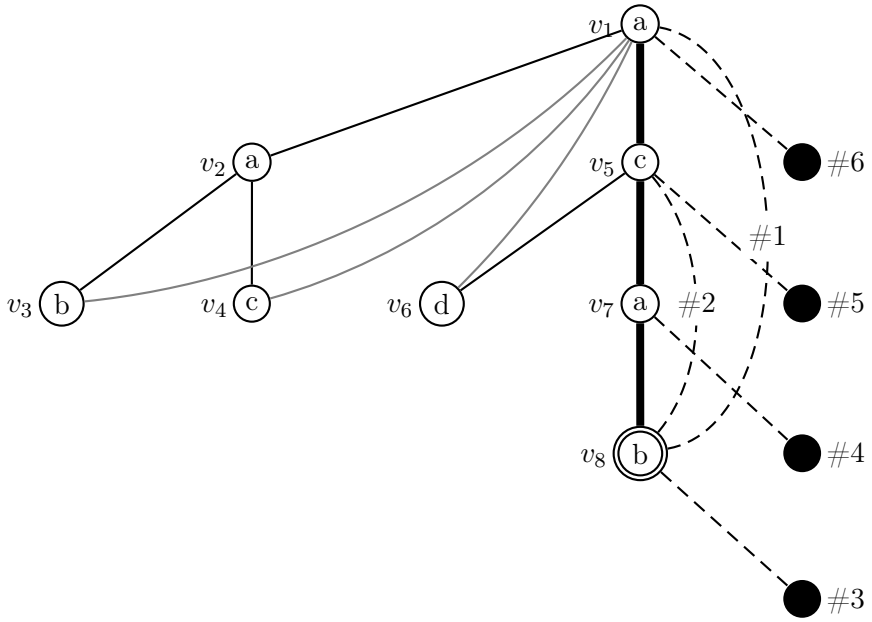


Figure 11.5: Rightmost Path Extensions. The bold path is the rightmost path in the DFS tree. The *rightmost vertex* is v_8. Solid black lines indicate the *forward* edges, which are part of the DFS tree. The *backward* edges, which by definition are not part of the DFS tree, are shown in gray. The set of possible extensions on the rightmost path are shown with dashed lines. The precedence ordering of the extensions is also shown.

For generating new candidates from a given graph G, we extends 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 need to 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 v_r is the rightmost vertex, the extension (v_r, v_i) is tried before (v_r, v_j) if $i < j$. In other words, backward extensions closer to the root are considered before those further 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 further from the root are extended before those closer to the root.

Example 11.5: Consider the order of extensions shown in Figure 11.5. Node v_8 is the rightmost vertex, thus we have to try backward extensions from it. The first extension (#1) is the backward edge (v_8, v_1) connecting v_8 to the root, and the next extension (#2) is then (v_8, v_5). No other backward extensions are possible, without introducing multiple edges between the same pair of vertices. Among the forward edges, gSpan tries forward extensions in reverse order, starting from the rightmost vertex v_8 (extension #3) and ending at the root (extension #6). Thus, the forward extension (v_8, v_x) (#3) comes before (v_7, v_x) (#4), and so on.

11.2.2 Canonical Code

When generating candidates using rightmost path extensions, it is possible that duplicate, i.e., isomorphic, graphs are generated via different extensions. Among the isomorphic candidates, we need to keep only one for further extensions, whereas the others can be pruned to avoid redundant computations. The main idea is that if we can somehow sort or rank the isomorphic graphs, we can pick the *canonical representative*, say the one with the least rank, and only extend that graph. We do this via the notion of *canonical DFS code*.

Let G be a graph and let G_T is a DFS spanning tree for G. The DFS tree G_T 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, whereas the DFS edge ordering is obtained by following the edges between consecutive nodes in DFS order, with the 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 with a given DFS tree G_T, denoted DFScode(G_1), is then 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. That is, in the DFS code, each edge is augmented by the labels of the nodes incident with it, as well as the edge label.

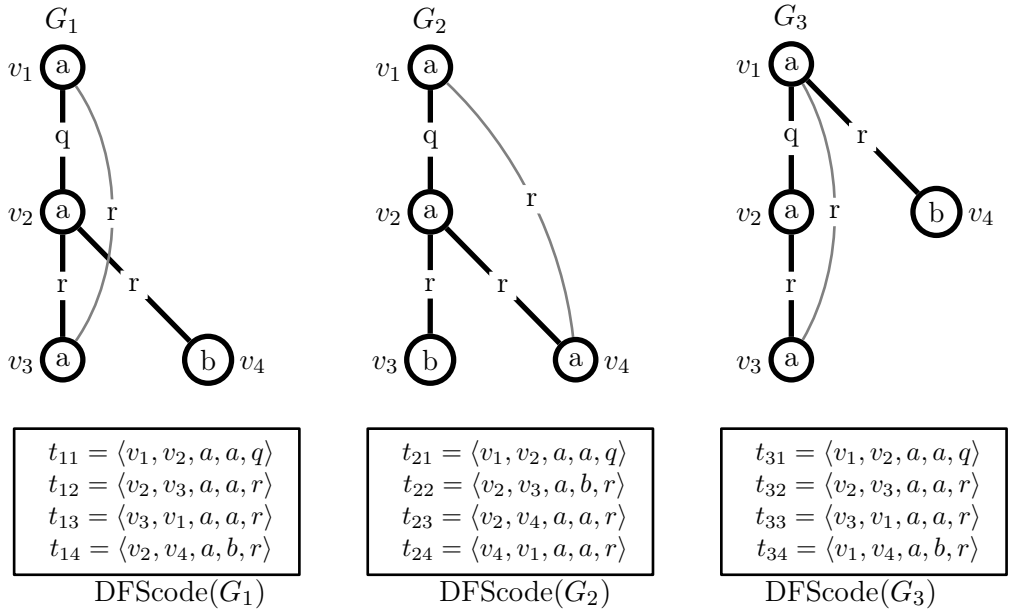


Figure 11.6: Canonical DFS Code. G_1 is canonical, whereas G_2 and G_3 are non-canonical. Vertex label set $\Sigma_V = \{a, b\}$, and edge label set $\Sigma_E = \{q, r\}$.

Example 11.6: Figure 11.6 shows the DFS codes for three graphs, which are all isomorphic to each other. The graphs have both node and edge labels drawn from the label sets $\Sigma_V = \{a, b\}$ and $\Sigma_E = \{q, r\}$. The edge labels are shown centered on the edges. The bold edges comprise the DFS tree for each graph. Consider the DFS code for G_1. The DFS node ordering is v_1, v_2, v_3, v_4, whereas the DFS edge ordering is (v_1, v_2), (v_2, v_3), (v_3, v_1), and (v_2, v_4). Based on the DFS edge ordering, the first tuple in the DFS code is therefore $\langle v_1, v_2, a, a, q \rangle$. The next tuple is $\langle v_2, v_3, a, a, r \rangle$ and so on. The DFS code for each graph is shown in the corresponding box below the graph.

Canonical DFS Code: A subgraph is *canonical* if it has the smallest DFS code among all possible isomorphic graphs, with the ordering between codes defined as follows. Let t_1 and t_2 be any two DFS code tuples

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

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

$$\begin{aligned} &\text{i) } (v_i, v_j) <_e (v_x, v_y), \text{or} \\ &\text{ii) } (v_i, v_j) = (v_x, v_y) \text{ and } \big\langle L(v_i), L(v_j), L(v_i, v_j) \big\rangle <_l \big\langle L(v_x), L(v_y), L(v_x, v_y) \big\rangle \end{aligned} \tag{11.2}$$

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. For example the label tuple $\langle a, a, r \rangle <_l \langle a, b, q \rangle$ since we have $L(v_i) = a = L(v_x)$, but $L(v_j) = a < b = L(v_y)$. The edge ordering $<_e$ is derived from the rules for rightmost path extension, namely that all of a node's backwards extensions must be considered before any forward edge from that node, and deep DFS trees are preferred over bushy DFS trees.

Formally, 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

i) If e_{ij} and e_{xy} are both forward edges, then a) $j < y$ or, b) $j = y$ and $i > x$. That is, a) a forward extension to a node earlier in the DFS node order is smaller, or b) if both the forward edges point to a node with the same DFS node order, then the forward extension from a node deeper in the tree (i.e., later in the DFS node order) is smaller.

ii) If e_{ij} and e_{xy} are both backward edges, then a) $i < x$ or b) $i = x$ and $j < y$. That is, a) a backward edge from a node earlier in the DFS node order is smaller, or b) if both the backward edges originate from the same node, then the backward edge to a node earlier in DFS node order (i.e., closer to the root along the rightmost path) is smaller.

iii) If e_{ij} is a forward and e_{xy} is a backward edge, then $j \leq x$. That is, a forward edge to a node earlier in the DFS node order is smaller than a backward edge from that node or any node that comes after it in DFS node order.

iv) If e_{ij} is a backward and e_{xy} is a forward edge, then $i < y$. That is, a backward edge from a node earlier in DFS node order is smaller than a forward edge to any later node.

Given any two DFS codes, we can compare them tuple by tuple to check which is smaller. In particular, the canonical DFS code for a graph G is defined as follows

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

Given a candidate subgraph G, we can first determine whether its DFS code is canonical or not. Only canonical graphs need to be retained for further processing, whereas non-canonical candidates can be removed from further consideration.

Example 11.7: Consider the DFS codes for the three graphs shown in Figure 11.6. Comparing G_1 and G_2, we find that $t_{11} = t_{21}$, but $t_{12} < t_{22}$, since $\langle a, a, r\rangle <_l \langle a, b, r\rangle$. Comparing the codes for G_1 and G_3, we find that the first three tuples are equal for both the graphs, but $t_{14} < t_{34}$, since $(v_2, v_4) <_e (v_1, v_4)$, due to condition i) above. That is, both are forward edges, and we have $v_j = v_4 = v_y$ with $v_i = v_2 > v_1 = v_x$. In fact, it can be shown that the code for G_1 is the canonical DFS code for all graphs isomorphic to G_1. Thus, G_1 is the canonical candidate.

11.3 The gSpan Algorithm

We consider the gSpan approach to mine all frequent subgraphs from a database of graphs. We detail below how it generates candidate patterns via edge extensions along the rightmost path, the canonical coding it uses to detect and eliminate duplicate candidates, and finally how it does support counting via subgraph isomorphism checks against the database.

Given a database $\mathbf{D} = \{G_1, G_2, \cdots, G_n\}$ comprising n graphs, and given a minimum support threshold *minsup*, the goal of graph mining is to enumerate all (connected) subgraphs G that are frequent, i.e., $\text{sup}(G) \geq minsup$. In gSpan, each graph is represented by its canonical DFS code, so that the task of enumerating frequent subgraphs is equivalent to the task of generating all canonical DFS codes for frequent subgraphs. Algorithm 11.1 shows the pseudo-code for gSpan.

Algorithm 11.1: Algorithm gSpan

```
   // Initial Call: gSpan (∅, D, minsup)
   gSpan (C, D, minsup):
1  E ← RightMostPath-Extensions(C, D) // extensions and supports
2  foreach ⟨t, sup(t)⟩ ∈ E do
3      C' ← C ∪ t // extend the code with extended edge tuple t
4      sup(C') ← sup(t) // record the support of new extension
       // recursively call gSpan if code is frequent and canonical
5      if sup(C') ≥ minsup and IsCanonical (C') then
6          gSpan (C', D, minsup)
```

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 (Line 1). The function RightMostPath-Extensions returns the set of edge extensions along with their supports $\mathcal{E}$. Each

extended edge t in $\mathcal{E}$ leads to a new candidate DFS code $C' = C \cup \{t\}$, with support $sup(C) = sup(t)$ (Lines 3-4). For each new candidate code, gSpan checks whether is it frequent and canonical, and if so gSpan recursively extends C' (Lines 5-6). The algorithm stops when there are no more frequent and canonical extensions possible.

Example 11.8: Consider the example graph database comprising G_1 and G_2 shown in Figure 11.7. Let $minsup = 2$, i.e., assume that we are interested in mining subgraphs that appear in both the graphs in the database. For each graph the node labels and node numbers are both shown, e.g., the node a^{10} in G_1 means that node 10 has label a.

Figure 11.8 shows the candidate patterns enumerated by gSpan. For each candidate the nodes are numbered in the DFS tree order. The solid boxes show frequent subgraphs, whereas the dotted boxes show the infrequent ones. The dashed boxes represent non-canonical codes. Subgraphs that do not occur even once are not shown. The figure shows the DFS codes and their corresponding subgraphs.

The mining process begins with the empty DFS code C_0 corresponding to the empty subgraph. The set of possible 1-edge extensions comprise the new set of candidates. Among these, C_3 is pruned since it is not canonical (it is isomorphic to C_2), whereas C_4 is pruned since it is not frequent. The remaining two candidates C_1 and C_2 are both frequent and canonical, and are thus considered for further extension. The depth-first search considers C_1 before C_2, with the rightmost path extensions of C_1 being C_5 and C_6. However, C_6 is not canonical (it is isomorphic to C_5, which has the canonical DFS code). Further extensions of C_5 are processed recursively. Once the recursion from C_1 completes, gSpan moves on to C_2 which will be recursively extended via rightmost edge extensions as illustrated by the subtree under C_2. After processing C_2 gSpan terminates since no other frequent and canonical extensions are found. In this example, C_{12} is a maximal frequent graph, i.e., no supergraph of C_{12} is frequent.

This example also shows the importance of duplicate elimination via canonical checking. The groups of isomorphic subgraphs encountered during the execution of gSpan are as follows: $\{C_2, C_3\}$, $\{C_5, C_6, C_{17}\}$, $\{C_7, C_{19}\}$, $\{C_9, C_{25}\}$, $\{C_{20}, C_{21}, C_{22}, C_{24}\}$, and $\{C_{12}, C_{13}, C_{14}\}$. Within each group the first graph is canonical and thus the remaining codes are pruned.

For a complete description of gSpan we have to specify the algorithms for enumerating the rightmost path extensions and their support, so that infrequent patterns can be eliminated, and for checking whether a given DFS code is canonical, so that duplicate patterns can be pruned. These are detailed below.

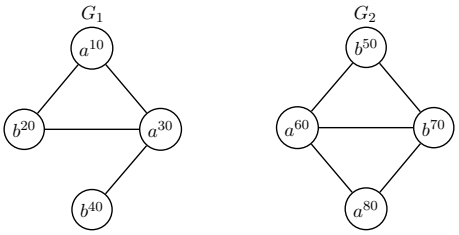


Figure 11.7: Example Graph Database

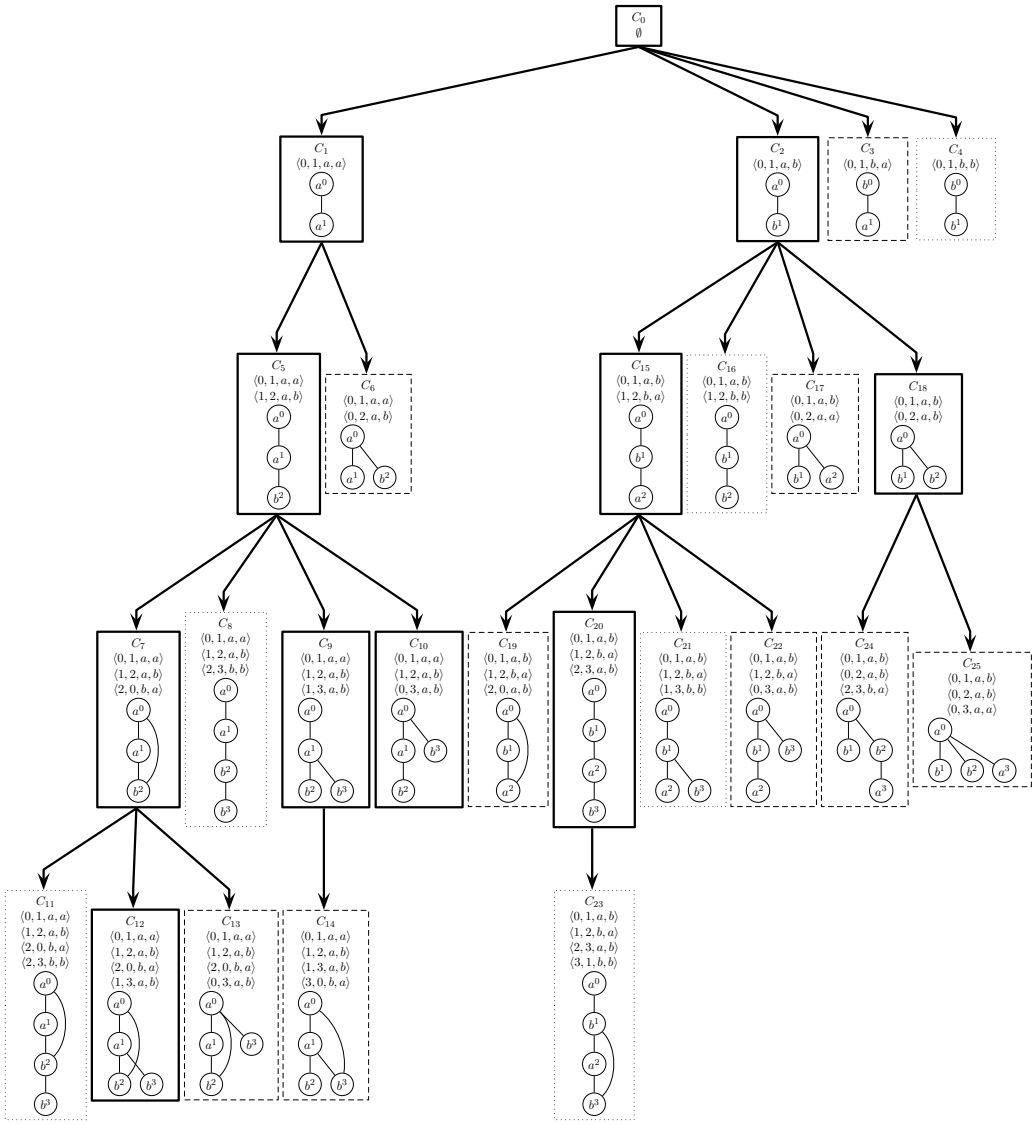


Figure 11.8: Frequent Graph Mining: $minsup = 2$. Solid boxes indicate frequent subgraphs, dotted indicate infrequent subgraphs, and dashed ones represent non-canonical codes.

11.3.1 Extension and Support Computation

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

Assume that $\mathbf{D} = \{G_1, G_2, \cdots, G_n\}$ comprises n graphs. Let $C = \{t_1, t_2, \cdots, t_k\}$ 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, which is accomplished via the pseudo-code in Algorithm 11.2.

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 $\langle 0, 1, L(x), L(y), L(x,y)\rangle$ (Lines 6-8). 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 via the function SubgraphIsomorphisms (Line 10). Given an isomorphism $\phi \in \Phi_i$, gSpan finds all possible forward and backward edge extensions, and stores them in the extension set $\mathcal{E}$.

Backward extensions (Lines 12-15) 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, the 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 that extension. Forward extensions (Lines 16-19) are allowed only from nodes on the rightmost path R to new nodes. For each node u in R, the algorithm finds its neighbors x in G_i that have not already been mapped to 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. Since a forward extension adds a new vertex to the graph $G(C)$, the id of the new node in C must be $u_r + 1$, i.e., one more than the highest numbered node in C, which by definition is the rightmost child u_r.

Once all the backward and forward extensions have been cataloged over all graphs G_i in the database $\mathbf{D}$, the method computes their support by counting the number of distinct graph ids that contribute to each extension. Finally, the method returns the set of all extensions and their supports in sorted order (increasing) based on the tuple comparison operator in (11.2).

Example 11.9: Consider the canonical code C and the corresponding graph $G(C)$ shown in Figure 11.9a. For this code all the vertices are on the rightmost path, i.e., $R = \{0, 1, 2\}$, and the rightmost child is $u_r = 2$.

All the possible isomorphisms from C to graphs G_1 and G_2 in the database

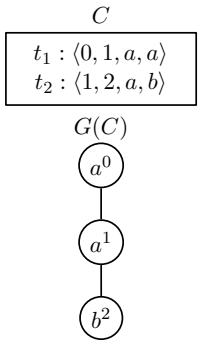
Algorithm 11.2: Rightmost Path Extensions and their Support

```
RightMostPath-Extensions (C, D):
R ← nodes on the rightmost path in C
u_r ← rightmost child in C // dfs number
E ← ∅ // set of extensions from C
foreach G_i ∈ D, i = 1, ··· , n do
    if C = ∅ then
        // add distinct label tuples in G_i as forward extensions
        foreach ⟨L(x), L(y), L(x, y)⟩ ∈ G_i do
            f = ⟨0, 1, L(x), L(y), L(x, y)⟩
            Add tuple f to E along with graph id i
    else
        Φ_i = SubgraphIsomorphisms(C, G_i)
        foreach isomorphism φ ∈ Φ_i do
            // backward extensions from rightmost child
            foreach x ∈ N_{G_i}(φ(u_r)) such that ∃v ← φ^{-1}(x) do
                if v ∈ R and (u_r, v) ∉ G(C) then
                    b = ⟨u_r, v, L(u_r), L(v), L(u_r, v)⟩
                    Add tuple b to E along with graph id i
            // forward extensions from nodes on rightmost path
            foreach u ∈ R do
                foreach x ∈ N_{G_i}(φ(u)) and ∄φ^{-1}(x) do
                    f = ⟨u, u_r + 1, L(φ(u)), L(x), L(φ(u), x)⟩
                    Add tuple f to E along with graph id i
// Compute the support of each extension
foreach extension s ∈ E do
    sup(s) = number of distinct graph ids that support tuple s
return set of pairs ⟨s, sup(s)⟩ for extensions s ∈ E, in tuple sorted order
```

(shown in Figure 11.7) are listed in Figure 11.9b as Φ_1 and Φ_2. For example, the first isomorphism $\phi : G(C) \rightarrow G_1$ is defined as

$$\phi(0) = 10 \qquad \phi(1) = 30 \qquad \phi(2) = 20$$

The list of possible backward and forward extensions for each isomorphism are shown in Figure 11.9c. For example, there are two possible edge extensions from the isomorphism ϕ given above. The first is a backward edge extension $\langle 2, 0, b, a \rangle$, since $(20, 10)$ is a valid back edge in G_1. That is, the node $x = 10$ is a neighbor



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

	0	1	2
Φ_1	10	30	20
	10	30	40
	30	10	20
Φ_2	60	80	70
	80	60	50
	80	60	70

(b) Subgraph Isomorphisms

id	extensions
G_1	$\{\langle 2, 0, b, a \rangle, \langle 1, 3, a, b \rangle\}$
	$\{\langle 1, 3, a, b \rangle, \langle 0, 3, a, b \rangle\}$
	$\{\langle 2, 0, b, a \rangle, \langle 0, 3, a, b \rangle\}$
G_2	$\{\langle 2, 0, b, a \rangle, \langle 2, 3, b, b \rangle, \langle 0, 3, a, b \rangle\}$
	$\{\langle 2, 3, b, b \rangle, \langle 1, 3, a, b \rangle\}$
	$\{\langle 2, 0, b, a \rangle, \langle 2, 3, b, b \rangle, \langle 1, 3, a, b \rangle\}$

(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

Figure 11.9: Rightmost Path Extensions

of $\phi(2) = 20$ in G_1, $\phi^{-1}(10) = 0 = v$ is on the rightmost path, and the edge $(2, 0)$ is not already in $G(C)$. The second extension is a forward one $\langle 1, 3, a, b \rangle$, since $(30, 40, a, b)$ is a valid extended edge in G_1. That is, $x = 40$ is a neighbor of $\phi(1) = 30$ in G_1, and node 40 has not already been mapped to any node in $G(C)$.

Given the set of all the edge extensions, and the graph ids that contribute to them, we obtain support for each distinct extension by counting how many distinct graph contribute to it. The final set of extensions, in sorted order, along with their support values is shown in Figure 11.9d. With $minsup = 2$, the only infrequent extension is $\langle 2, 3, b, b \rangle$.

Subgraph Isomorphisms: The key step in listing the edge extensions for a given code C is to enumerate all the possible isomorphisms from C to each graph $G_i \in \mathbf{D}$. The function SubgraphIsomorphisms, shown in Algorithm 11.3, accepts a code C and a graph G, an 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, if $L(x) = L(0)$ (Line 1). The method considers each tuple t_i in C and extends the current set of partial isomorphisms.

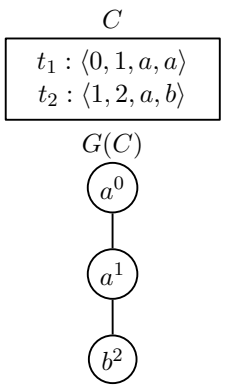
Algorithm 11.3: Enumerate subgraph isomorphisms

```
SubgraphIsomorphisms (C = {t_1, t_2, ..., t_k}, G):
1  Φ ← {φ(0) → x | x ∈ G and L(x) = L(0)}
2  foreach t_i ∈ C, i = 1, ..., k do
3      ⟨u, v, L(u), L(v), L(u, v)⟩ ← t_i // expand extended edge t_i
4      Φ' ← ∅ // partial isomorphisms including t_i
5      foreach partial isomorphism φ ∈ Φ do
6          if v > u then
               // forward edge
7              foreach x ∈ N_G(φ(u)) do
8                  if ∄φ^{-1}(x) and L(x) = L(v) and L(φ(u), x) = L(u, v) then
9                      φ' ← φ ∪ {φ(v) → x}
10                     Add φ' to Φ
11         else
               // backward edge
12             if φ(v) ∈ N_{G_j}(φ(u)) then Add φ to Φ' // valid isomorphism
13     Φ ← Φ' // update partial isomorphisms
14 return Φ
```

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 (Lines 5-12. 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 (i.e., $\phi^{-1}(x)$ should not exist), and the node and edge labels should match (i.e., $L(x) = L(v)$, and $L(\phi(u), x) = L(u, v)$. If so, ϕ can be extended with the mapping $\phi(v) = x$. 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'. Thus, only those isomorphism that can be extended in the forward case, or those that satisfy the backward edge, are retained for further further processing. Once all the extended edges in C have been processed, the set Φ contains all the valid isomorphisms from C to G.

Example 11.10: Figure 11.10 illustrates the subgraph isomorphism enumeration algorithm from the code C to each of the graphs G_1 and G_2 in the database shown in Figure11.7.

For G_1, the set of isomorphisms Φ is initialized by mapping the first node of C to all nodes labeled a in G_1, since $L(0) = a$. Thus, $\Phi = \{\phi_1(0) \to 10, \phi_2(0) \to 30\}$. We next consider tuple in C, and see which isomorphisms can be extended. The first tuple $t_1 = \langle 0, 1, a, a\rangle$ is a forward edge, thus for ϕ_1, we consider neighbors x of 10



Initial Φ

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

Add t_1

id	ϕ	0, 1
G_1	ϕ_1	10, 30
	ϕ_2	30, 10
G_2	ϕ_1	60, 80
	ϕ_2	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_1	60, 80, 70
	ϕ'_2	80, 60, 50
	ϕ''_2	80, 60, 70

Figure 11.10: Subgraph Isomorphisms

that are labeled a and not included in the isomorphism yet. The only other vertex that satisfies this condition is 30, thus the isomorphism is extended by mapping $\phi_1(1) \rightarrow 30$. In a similar manner the second isomorphism ϕ_2 is extended by adding $\phi_2(1) \rightarrow 10$, as shown in Figure 11.10. For the second tuple $t_2 = \langle 1, 2, a, b \rangle$, the isomorphism ϕ_1 has two possible extensions, since 30 has two neighbors labeled b, namely 20 and 40. The extended mappings are denoted ϕ'_1 and ϕ''_1. For ϕ_2 there is only one extension.

The isomorphisms of C in G_2 can be found in a similar manner. The complete set of isomorphisms in each database graph are shown in Figure 11.10.

11.3.2 Canonicality Checking

Given a DFS code $C = \{t_1, t_2, \cdots, t_k\}$ 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 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 (11.2). 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. The pseudo-code for canonicality checking is given in Algorithm 11.4. The method can be considered as a restricted version of gSpan in that the graph $G(C)$ plays the role of a graph in the database, and C^* plays the role of a candidate extension. The key difference is that we consider only the smallest rightmost path edge extension among all the possible candidates.

Algorithm 11.4: Canonicality Checking: Algorithm IsCanonical

```
IsCanonical (C):
1 D_C ← {G(C)} // graph corresponding to code C
2 C* ← ∅ // initialize canonical DFScode
3 for i = 1 ··· k do
4     E = RightMostPath-Extensions(C*, D_C) // extensions of C*
5     ⟨s_i, sup(s_i)⟩ ← min{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* ← C* ∪ s_i
9 return true // no smaller code exists; C is canonical
```

Example 11.11: Consider the subgraph candidate C_{24} from Figure 11.8, which is replicated as graph G in Figure 11.11, along with its DFS code C. From an initial canonical code $C^* = \emptyset$, the smallest rightmost edge extension s_1 is added in Step 1. Since $s_1 = t_1$, we proceed to the next step, which finds the smallest edge extension s_2. Once again $s_2 = t_2$, so we proceed to the third step. The least possible edge extension for G^* is the extended edge s_3. However, we find that $s_3 < t_3$, which means that C cannot be canonical, and there is no need to try further edge extensions.

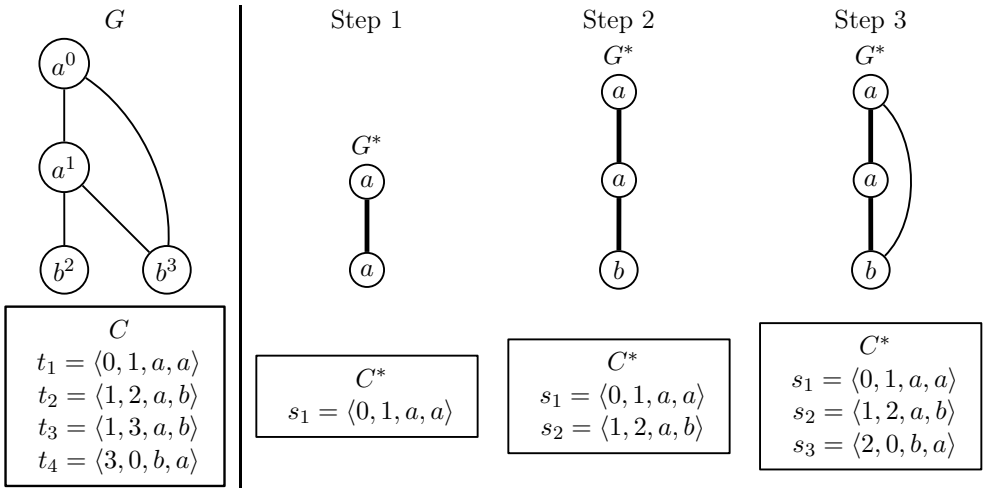


Figure 11.11: Canonicality Checking