

Chapter 8

Graph 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 interactions 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 they 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, a protein modules 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.

8.1 Foundations

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. We will write an edge $(x, y) \in E$ as xy , where both $x, y \in V$. If xy is an edge, we say that x and y are *adjacent* or that y is a *neighbor* of x . 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 $L(xy)$ to denote the label of the edge xy . Figure 8.1(a) shows an example of an unlabeled graph, whereas Figure 8.1(b) 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. In general $L(xy) \in \Sigma_E$, where Σ_E is the edge label set. We will assume from now on that all graphs are labeled.

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, such as the graph in Figure 8.2(a). In many applications of data mining, we are specifically interested in only *connected subgraphs*, i.e., when $V' \subseteq V$, $E' \subseteq E$, and for any $x, y \in V'$, there exists a *path* from x to y . Figure 8.2(b) shows a connected subgraph.

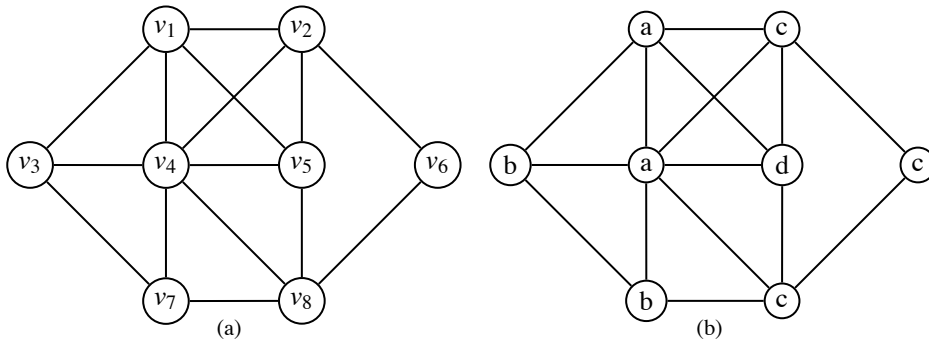


Figure 8.1: An unlabeled (a) and labeled (b) graph with 8 vertices

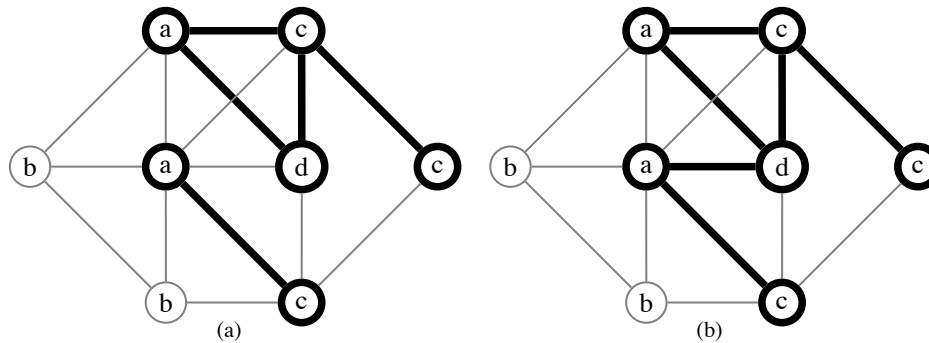


Figure 8.2: A subgraph (a) and connected subgraph (b)

Isomorphism: A graph $G' = (V', E')$ is said to be *isomorphic* to $G = (V, E)$ if there is a bijective function $\phi : V' \rightarrow V$, 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. We say that G' is isomorphic to a subgraph of G , or just *subgraph isomorphic* to G if the function ϕ is only injective but not surjective. For example, in Figure 8.3 G_1 and G_2 are isomorphic graphs, and G_3 is subgraph isomorphic to both G_1 and G_2 . If G' is subgraph isomorphic to G , we denote it as $G' \subseteq G$, and we also say that G contains G' .

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

$$sup(G) = |\{G_i \in \mathcal{D} \mid G \subseteq G_i\}| \quad (8.1)$$

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 subgraphs with $sup(G) \geq minsup$. We assume that all subgraphs are connected.

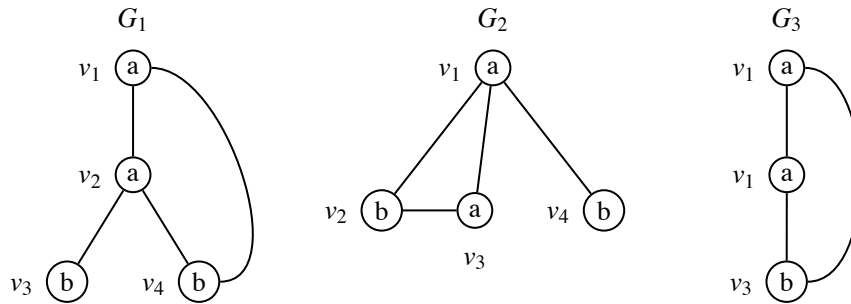


Figure 8.3: Graph and Subgraph Isomorphism

8.2 Subgraph Mining

To mine all the frequent subgraphs, we have to search over the space of all possible graph patterns, which is enormously large. 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 to exclude each of the $O(m^2)$ edges. Many of these will not be connected, but we can still use this worst case bound. When we add labels to the vertices and edges, the number of labeled graphs will be even more. Assume that $|\Sigma_V| = |\Sigma_E| = l$, then there are l^m possible ways to label the vertices and there are l^{m^2} ways to label the edges. Thus the number of possible labeled subgraphs with m vertices is $2^{O(m^2)} l^m l^{m^2} = O((2 \cdot l)^{m^2})$. This is the worst case pessimistic 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 graph.

There are two main challenges in frequent subgraph mining. The first is to systematically generate candidate subgraphs in a non-redundant manner, such that we do not generate the same graph 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 precisely the set of graphs that contain a given candidate.

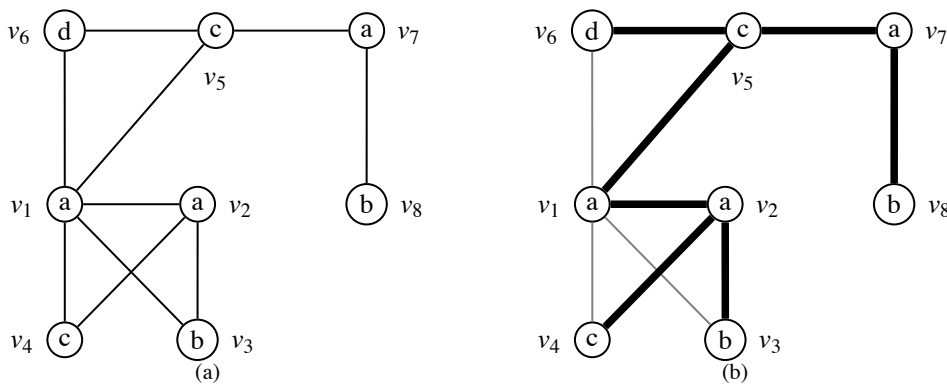


Figure 8.4: A graph (a) and its depth-first spanning tree (b)

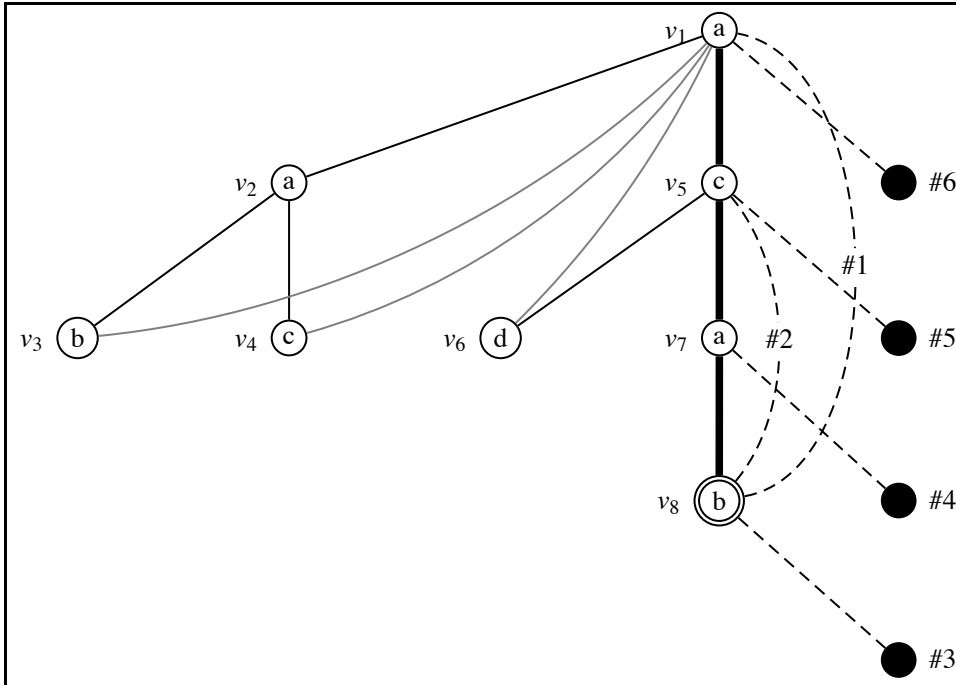


Figure 8.5: Rightmost Extensions. The bold path is the rightmost path in the DFS tree. The *rightmost vertex* is v_8 . Solid black lines indicate the *forward edges*, which are part of the DFS tree. The *backward edges* not part of the DFS tree, are shown in gray. The set of possible extensions on the rightmost path are shown with dashed lines. The precedence ordering of the extensions is also shown.

8.2.1 Systematic Candidate Generation: Rightmost Path Extension

As mentioned above, we only mine connected subgraphs. We will use *edge-growth* as the basic mechanism for extending the candidates. The mining process will proceed in a level-wise manner, starting with subgraphs with zero edges. As the next level, we will consider all subgraphs with one edge. At level two, we consider all graphs having two edges, and so on.

We will enumerate subgraph patterns using a strategy called the *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. For example, consider the graph shown in Figure 8.4(a). One of the possible DFS spanning trees is shown in Figure 8.4(b), which was obtained by starting at v_1 and then choosing the vertex with the smallest index at each step. Figure 8.5 shows the DFS tree of Figure 8.4(b) in a rearranged manner to emphasize its tree structure. 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). The rightmost path is shown in bold in Figure 8.5.

For generating new candidates from a given graph G , we always 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. Backward extension does not add a new vertex, whereas a forward extension adds a new vertex. Note also that there is an ordering on the

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

Here $<_e$ 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 tuple $\langle a, a, r \rangle < \langle a, b, q \rangle$ since if we compare the two tuples in an element-wise manner, we find that $L(v_j) = a < b = L(v_y)$.

The edge order $<_e$ is more involved. Let $e_{ij} = \langle v_i, v_j \rangle$ and $e_{xy} = \langle v_x, v_y \rangle$. We write $e_{ij} <_e e_{xy}$ if the follow conditions are met:

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

Given the DFS codes for any two graphs, we can compare them tuple by tuple to check which is smaller. For example, consider the DFS codes for the three graphs shown in Figure 8.6. 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 G_1 with G_3 we find that the first three tuples are equal in both graphs, but $t_{14} < t_{34}$, since $\langle v_2, v_4 \rangle <_e \langle v_1, v_4 \rangle$. Applying the rules for edge ordering $<_e$, we find that condition i) applies, and $v_j = v_4 = v_y$ but $v_i = v_2 > v_1 = v_x$. In fact it can be shown that G_1 has the minimal DFS code among all graphs that are isomorphic to it. Thus G_1 is the canonical representative.

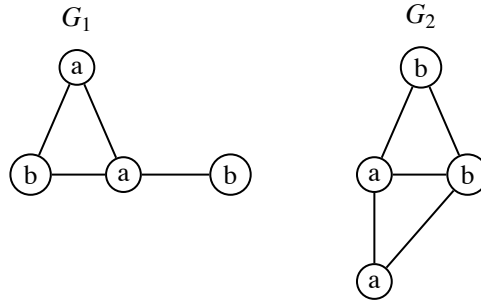


Figure 8.7: Example Graph Database

8.2.3 Subgraph Mining Algorithm

Having laid the foundation of non-redundant candidate enumeration and rightmost extensions, we are now ready to mine all possible subgraphs that are frequent. Consider the example graph database shown in Figure 8.7. Let $minsup = 2$, thus we are interested in mining the subgraphs that appear in both the graphs in the database.

The basic algorithm for mining all frequent (connected) subgraphs is as follows: Proceed in a level-wise manner, based on rightmost edge extensions. Check if the new candidate is canonical, if not prune it. If the candidate is canonical, then obtain its support in the database. If it is not frequent, prune it. On the other hand, if it is frequent, generate new extensions from it. The algorithm stops when there are no more candidate extensions possible. The above steps can also be applied in the depth-first manner, but we will illustrate the steps using a level-wise breadth-first approach.

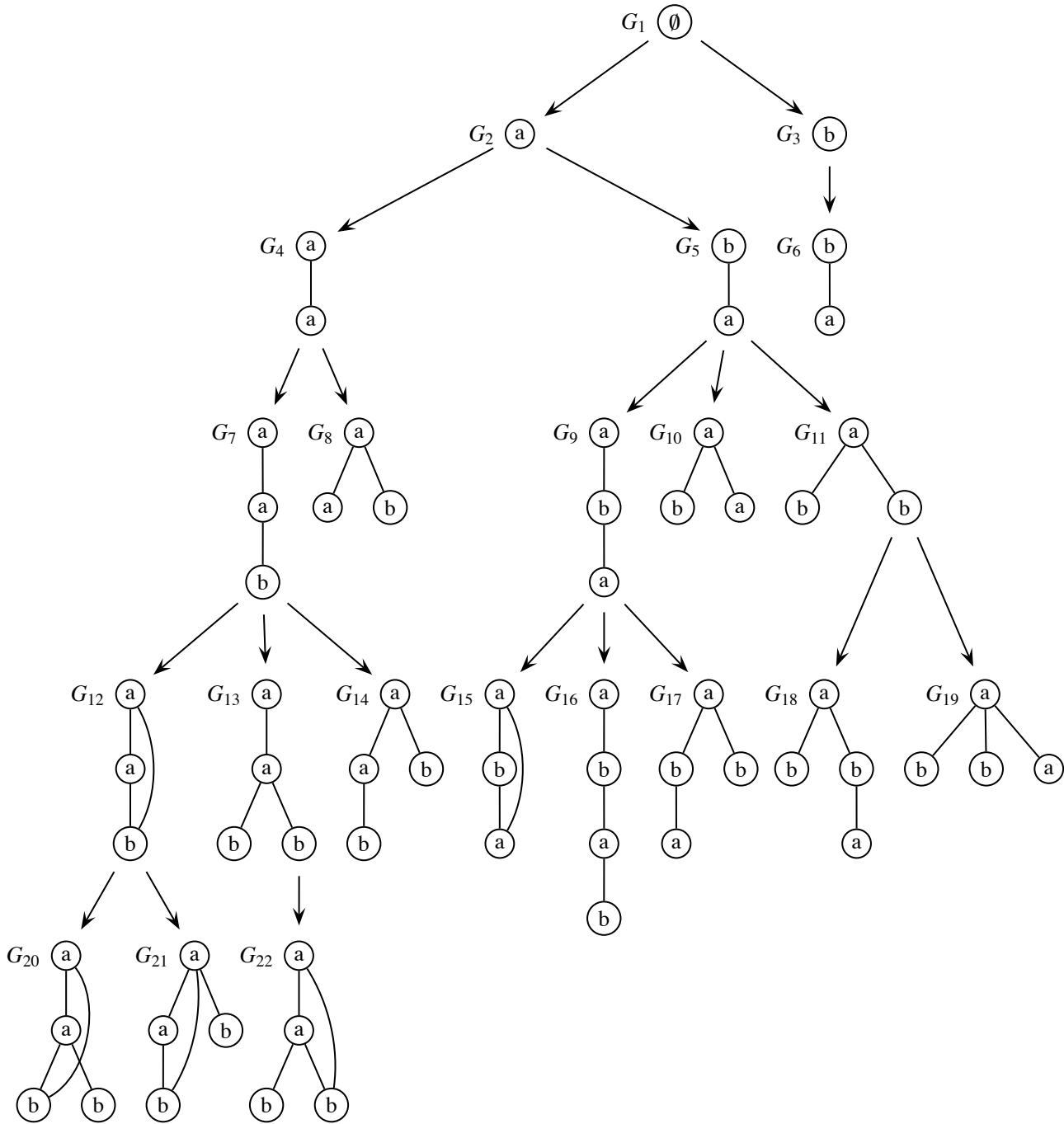


Figure 8.8: Frequent Graph Mining

Figure 8.8 shows the entire graph mining process. The infrequent graphs are not shown, but all non-canonical graphs are shown. The mining process begins with the single labels a and b . Next we proceed to the single edge graphs. Here we find that G_6 is isomorphic to G_4 and G_4 is canonical, so G_6 will not

be extended further. At level 2, G_8 and G_{10} will be pruned as they are isomorphic to G_7 , and G_7 is canonical. At level 3, the isomorphic groups of graphs in sorted order are $\{G_{12}, G_{15}\}$, $\{G_{13}, G_{19}\}$, $\{G_{14}\}$, and $\{G_{16}, G_{17}, G_{18}\}$. Thus only the canonical graphs G_{12} , G_{13} , G_{14} , and G_{16} will be further extended. Finally at level 4, we have one isomorphic group $\{G_{20}, G_{21}, G_{22}\}$, out of which G_{20} is canonical. This is also the maximal graph that is frequent with $minsup = 2$ in this example dataset.