9 Graph Mining, Social Network Analysis, and Multirelational Data Mining

We have studied frequent-itemset mining in Chapter 5 and sequential-pattern mining in Section 3 of Chapter 8. Many scientific and commercial applications need patterns that are more complicated than frequent itemsets and sequential patterns and require extra effort to discover. Such sophisticated patterns go beyond *sets* and *sequences*, toward *trees, lattices, graphs, networks*, and *other complex structures*.

As a general data structure, *graphs* have become increasingly important in modeling sophisticated structures and their interactions, with broad applications, including chemical informatics, bioinformatics, computer vision, video indexing, text retrieval, and Web analysis. *Mining frequent subgraph patterns* for further characterization, discrimination, classification, and cluster analysis becomes an important task. Moreover, graphs that link many nodes together may form different kinds of networks, such as telecommunication networks, computer networks, biological networks, and Web and social community networks. Because such networks have been studied extensively in the context of social networks, their analysis has often been referred to as *social network analysis*. Furthermore, in a relational database, objects are semantically linked across multiple relations. Mining in a relational database often requires mining across multiple interconnected relations, which is similar to mining in connected graphs or networks. Such kind of mining across data relations is considered *multirelational data mining*.

In this chapter, we study knowledge discovery in such interconnected and complex structured data. Section 9.1 introduces graph mining, where the core of the problem is mining frequent subgraph patterns over a collection of graphs. Section 9.2 presents concepts and methods for social network analysis. Section 9.3 examines methods for multirelational data mining, including both cross-relational classification and user-guided multirelational cluster analysis.

9.1 Graph Mining

Graphs become increasingly important in modeling complicated structures, such as circuits, images, chemical compounds, protein structures, biological networks, social

networks, the Web, workflows, and XML documents. Many graph search algorithms have been developed in chemical informatics, computer vision, video indexing, and text retrieval. With the increasing demand on the analysis of large amounts of structured data, graph mining has become an active and important theme in data mining.

Among the various kinds of graph patterns, *frequent substructures* are the very basic patterns that can be discovered in a collection of graphs. They are useful for characterizing graph sets, discriminating different groups of graphs, classifying and clustering graphs, building graph indices, and facilitating similarity search in graph databases. Recent studies have developed several graph mining methods and applied them to the discovery of interesting patterns in various applications. For example, there have been reports on the discovery of active chemical structures in HIV-screening datasets by contrasting the support of frequent graphs between different classes. There have been studies on the use of frequent structures as features to classify chemical compounds, on the frequent graph mining technique to study protein structural families, on the detection of considerably large frequent subpathways in metabolic networks, and on the use of frequent graph patterns for graph indexing and similarity search in graph databases. Although graph mining may include mining frequent subgraph patterns, graph classification, clustering, and other analysis tasks, in this section we focus on mining frequent subgraphs. We looks at various methods, their extensions, and applications.

9.1.1 Methods for Mining Frequent Subgraphs

Before presenting graph mining methods, it is necessary to first introduce some preliminary concepts relating to frequent graph mining.

We denote the **vertex set** of a graph g by $V(g)$ and the **edge set** by $E(g)$. A label function, L, maps a vertex or an edge to a label. A graph g is a **subgraph** of another graph g' if there exists a subgraph isomorphism from g to g'. Given a labeled graph data set, $D = \{G_1, G_2, \ldots, G_n\}$, we define $support(g)$ (or $frequency(g)$) as the percentage (or number) of graphs in D where g is a subgraph. A **frequent graph** is a graph whose support is no less than a minimum support threshold, *min_sup*.

Example 9.1 **Frequent subgraph.** Figure 9.1 shows a sample set of chemical structures. Figure 9.2 depicts two of the frequent subgraphs in this data set, given a minimum support of 66.6%. ■

"*How can we discover frequent substructures?*" The discovery of frequent substructures usually consists of two steps. In the first step, we generate frequent substructure candidates. The frequency of each candidate is checked in the second step. Most studies on frequent substructure discovery focus on the optimization of the first step, because the second step involves a subgraph isomorphism test whose computational complexity is excessively high (i.e., NP-complete).

In this section, we look at various methods for frequent substructure mining. In general, there are two basic approaches to this problem: an Apriori-based approach and a pattern-growth approach.

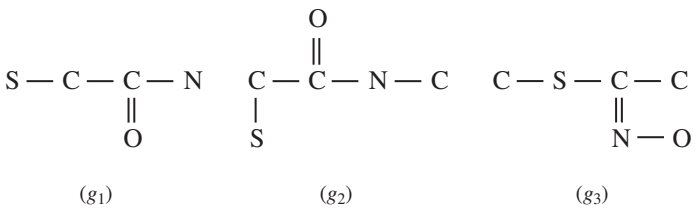

Figure 9.1 A sample graph data set.

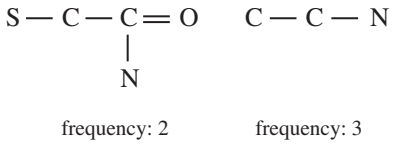

Figure 9.2 Frequent graphs.

Apriori-based Approach

Apriori-based frequent substructure mining algorithms share similar characteristics with Apriori-based frequent itemset mining algorithms (Chapter 5). The search for frequent graphs starts with graphs of small "size," and proceeds in a bottom-up manner by generating candidates having an extra vertex, edge, or path. The definition of graph size depends on the algorithm used.

The general framework of Apriori-based methods for frequent substructure mining is outlined in Figure 9.3. We refer to this algorithm as AprioriGraph. S_k is the frequent substructure set of size k. We will clarify the definition of graph size when we describe specific Apriori-based methods further below. AprioriGraph adopts a *level-wise* mining methodology. At each iteration, the size of newly discovered frequent substructures is increased by one. These new substructures are first generated by joining two similar but slightly different frequent subgraphs that were discovered in the previous call to AprioriGraph. This candidate generation procedure is outlined on line 4. The frequency of the newly formed graphs is then checked. Those found to be frequent are used to generate larger candidates in the next round.

The main design complexity of Apriori-based substructure mining algorithms is the candidate generation step. The candidate generation in frequent itemset mining is straightforward. For example, suppose we have two frequent itemsets of size-3: (*abc*) and (*bcd*). The frequent itemset candidate of size-4 generated from them is simply (*abcd*), derived from a join. However, the candidate generation problem in frequent substructure mining is harder than that in frequent itemset mining, because there are many ways to join two substructures.

Algorithm: AprioriGraph. Apriori-based frequent substructure mining.

Input:

- D, a graph data set;
- *min_sup*, the minimum support threshold.

Output:

- S_k, the frequent substructure set.

Method:

(1) $S_{k+1} \leftarrow \varnothing$;

(2) **for each** frequent $g_i \in S_k$ **do**

(3) **for each** frequent $g_j \in S_k$ **do**

(4) **for each** size $(k+1)$ graph g formed by the merge of g_i and g_j **do**

(5) **if** g is frequent in D and $g \notin S_{k+1}$ **then**

(6) insert g into S_{k+1};

(7) **if** $s_{k+1} \neq \varnothing$ **then**

(8) *AprioriGraph*(D, *min_sup*, S_{k+1});

(9) **return**;

Figure 9.3 AprioriGraph.

Recent Apriori-based algorithms for frequent substructure mining include AGM, FSG, and a path-join method. AGM shares similar characteristics with Apriori-based itemset mining. FSG and the path-join method explore edges and connections in an Apriori-based fashion. Each of these methods explores various candidate generation strategies.

The AGM algorithm uses a *vertex-based candidate generation* method that increases the substructure size by one vertex at each iteration of AprioriGraph. Two size-k frequent graphs are joined only if they have the same size-$(k-1)$ subgraph. Here, *graph size* is the number of vertices in the graph. The newly formed candidate includes the size-$(k-1)$ subgraph in common and the additional two vertices from the two size-k patterns. Because it is undetermined whether there is an edge connecting the additional two vertices, we actually can form two substructures. Figure 9.4 depicts the two substructures joined by two chains (where a chain is a sequence of connected edges).

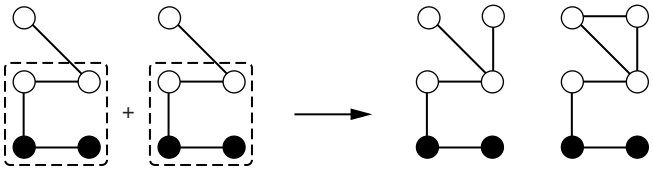

Figure 9.4 AGM: Two substructures joined by two chains.

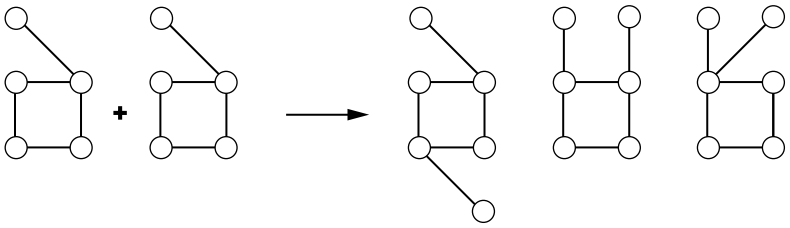

Figure 9.5 FSG: Two substructure patterns and their potential candidates.

The FSG algorithm adopts an *edge-based candidate generation* strategy that increases the substructure size by one edge in each call of AprioriGraph. Two size-k patterns are merged if and only if they share the same subgraph having $k-1$ edges, which is called the **core**. Here, *graph size* is taken to be the number of edges in the graph. The newly formed candidate includes the core and the additional two edges from the size-k patterns. Figure 9.5 shows potential candidates formed by two structure patterns. Each candidate has one more edge than these two patterns. This example illustrates the complexity of joining two structures to form a large pattern candidate.

In a third Apriori-based approach, an *edge-disjoint path method* was proposed, where graphs are classified by the number of disjoint paths they have, and two paths are edge-disjoint if they do not share any common edge. A substructure pattern with $k+1$ disjoint paths is generated by joining substructures with k disjoint paths.

Apriori-based algorithms have considerable overhead when joining two size-k frequent substructures to generate size-$(k+1)$ graph candidates. In order to avoid such overhead, non-Apriori-based algorithms have recently been developed, most of which adopt the pattern-growth methodology. This methodology tries to extend patterns directly from a single pattern. In the following, we introduce the pattern-growth approach for frequent subgraph mining.

Pattern-Growth Approach

The Apriori-based approach has to use the breadth-first search (BFS) strategy because of its level-wise candidate generation. In order to determine whether a size-$(k+1)$ graph is frequent, it must check all of its corresponding size-k subgraphs to obtain an upper bound of its frequency. Thus, before mining any size-$(k+1)$ subgraph, the Apriori-like

Algorithm: PatternGrowthGraph. Simplistic pattern growth-based frequent substructure mining. **Input**:

- g, a frequent graph;
- D, a graph data set;
- *min_sup*, minimum support threshold.

Output:

- The frequent graph set S.

Method:

(1) if $g \in S$ then return;

(2) **else** insert g into S;

(3) scan D once, find all the edges e such that g can be extended to $g \diamond_x e$;

(4) **for each** frequent $g \diamond_x e$ **do**

(5) *PatternGrowthGraph*($g \diamond_x e$, D, *min_sup*, S);

(6) **return**;

Figure 9.6 PatternGrowthGraph.

approach usually has to complete the mining of size-k subgraphs. Therefore, BFS is necessary in the Apriori-like approach. In contrast, the *pattern-growth approach* is more flexible regarding its search method. It can use breadth-first search as well as depth-first search (DFS), the latter of which consumes less memory.

A graph g can be *extended* by adding a new edge e. The newly formed graph is denoted by $g \diamond_x e$. Edge e may or may not introduce a new vertex to g. If e introduces a new vertex, we denote the new graph by $g \diamond_{xf} e$, otherwise, $g \diamond_{xb} e$, where f or b indicates that the extension is in a *forward* or *backward* direction.

Figure 9.6 illustrates a general framework for pattern growth–based frequent substructure mining. We refer to the algorithm as PatternGrowthGraph. For each discovered graph g, it performs extensions recursively until all the frequent graphs with g embedded are discovered. The recursion stops once no frequent graph can be generated.

PatternGrowthGraph is simple, but not efficient. The bottleneck is at the inefficiency of extending a graph. The same graph can be discovered many times. For example, there may exist n different $(n-1)$-edge graphs that can be extended to the same n-edge graph. The repeated discovery of the same graph is computationally inefficient. We call a graph that is discovered second time a **duplicate graph**. Although line

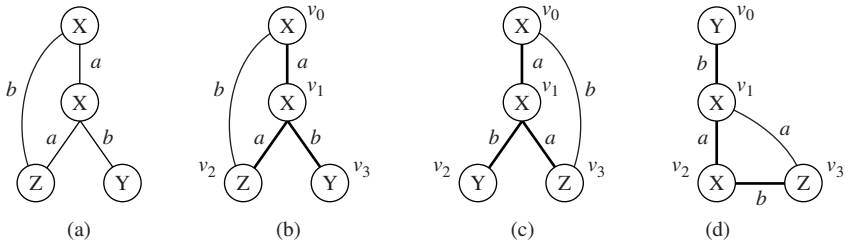

Figure 9.7 DFS subscripting.

1 of PatternGrowthGraph gets rid of duplicate graphs, the generation and detection of duplicate graphs may increase the workload. In order to reduce the generation of duplicate graphs, each frequent graph should be extended as conservatively as possible. This principle leads to the design of several new algorithms. A typical such example is the gSpan algorithm as described below.

The gSpan algorithm is designed to reduce the generation of duplicate graphs. It need not search previously discovered frequent graphs for duplicate detection. It does not extend any duplicate graph, yet still guarantees the discovery of the complete set of frequent graphs.

Let's see how the gSpan algorithm works. To traverse graphs, it adopts depth-first search. Initially, a starting vertex is randomly chosen and the vertices in a graph are marked so that we can tell which vertices have been visited. The visited vertex set is expanded repeatedly until a full depth-first search (DFS) tree is built. One graph may have various DFS trees depending on how the depth-first search is performed (i.e., the vertex visiting order). The darkened edges in Figure 9.7(b) to 9.7(d) show three DFS trees for the same graph of Figure 9.7(a). The vertex labels are x, y, and z; the edge labels are a and b. Alphabetic order is taken as the default order in the labels. When building a DFS tree, the visiting sequence of vertices forms a linear order. We use subscripts to record this order, where $i < j$ means v_i is visited before v_j when the depth-first search is performed. A graph G subscripted with a DFS tree T is written as G_T. T is called a **DFS subscripting** of G.

Given a DFS tree T, we call the starting vertex in T, v_0, the *root*. The last visited vertex, v_n, is called the *right-most vertex*. The straight path from v_0 to v_n is called the *right-most path*. In Figure 9.7(b) to 9.7(d), three different subscriptings are generated based on the corresponding DFS trees. The right-most path is (v_0, v_1, v_3) in Figure 9.7(b) and 9.7(c), and (v_0, v_1, v_2, v_3) in Figure 9.7(d).

PatternGrowth extends a frequent graph in every possible position, which may generate a large number of duplicate graphs. The gSpan algorithm introduces a more sophisticated extension method. The new method restricts the extension as follows: Given a graph G and a DFS tree T in G, a new edge e can be added between the right-most vertex and other vertices on the right-most path (*backward extension*); or it can introduce a new vertex and connect to vertices on the right-most path (*forward extension*). Because both

kinds of extensions take place on the right-most path, we call them *right-most extension*, denoted by $G \diamond_r e$ (for brevity, T is omitted here).

Example 9.2 **Backward extension and forward extension.** If we want to extend the graph in Figure 9.7(b), the backward extension candidates can be (v_3, v_0). The forward extension candidates can be edges extending from v_3, v_1, or v_0 with a new vertex introduced. ■

Figure 9.8(b) to 9.8(g) shows all the potential right-most extensions of Figure 9.8(a). The darkened vertices show the right-most path. Among these, Figure 9.8(b) to 9.8(d) grows from the right-most vertex while Figure 9.8(e) to 9.8(g) grows from other vertices on the right-most path. Figure 9.8(b.0) to 9.8(b.4) are children of Figure 9.8(b), and Figure 9.8(f.0) to 9.8(f.3) are children of Figure 9.8(f). In summary, backward extension only takes place on the right-most vertex while forward extension introduces a new edge from vertices on the right-most path.

Because many DFS trees/subscriptings may exist for the same graph, we choose one of them as the *base subscripting* and only conduct right-most extension on that DFS tree/subscripting. Otherwise, right-most extension cannot reduce the generation of duplicate graphs because we would have to extend the same graph for every DFS subscripting.

We transform each subscripted graph to an edge sequence, called a **DFS code**, so that we can build an order among these sequences. The goal is to select the subscripting that generates the minimum sequence as its base subscripting. There are two kinds of orders in this transformation process: (1) *edge order*, which maps edges in a subscripted graph

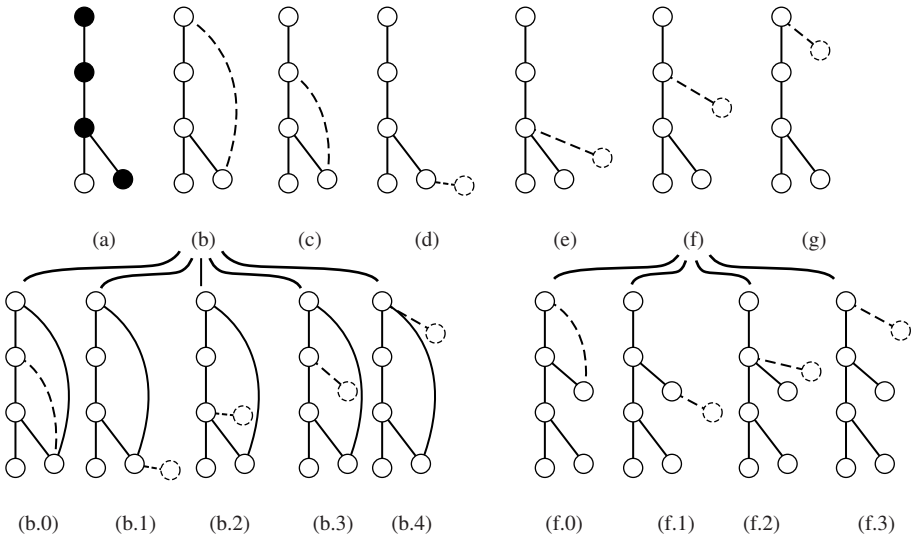

Figure 9.8 Right-most extension.

into a sequence; and (2) *sequence order*, which builds an order among edge sequences (i.e., graphs).

First, we introduce edge order. Intuitively, DFS tree defines the discovery order of forward edges. For the graph shown in Figure 9.7(b), the forward edges are visited in the order of $(0,1),(1,2),(1,3)$. Now we put backward edges into the order as follows. Given a vertex v, all of its backward edges should appear just before its forward edges. If v does not have any forward edge, we put its backward edges after the forward edge, where v is the second vertex. For vertex v_2 in Figure 9.7(b), its backward edge $(2,0)$ should appear after $(1,2)$ because v_2 does not have any forward edge. Among the backward edges from the same vertex, we can enforce an order. Assume that a vertex v_i has two backward edges, (i,j_1) and (i,j_2). If $j_1 < j_2$, then edge (i,j_1) will appear before edge (i,j_2). So far, we have completed the ordering of the edges in a graph. Based on this order, a graph can be transformed into an edge sequence. A complete sequence for Figure 9.7(b) is $(0,1),(1,2),(2,0),(1,3)$.

Based on this ordering, three different DFS codes, γ_0, γ_1, and γ_2, generated by DFS subscriptings in Figure 9.7(b), 9.7(c), and 9.7(d), respectively, are shown in Table 9.1. An edge is represented by a 5-tuple, $(i,j,l_i,l_{(i,j)},l_j)$, l_i and l_j are the labels of v_i and v_j, respectively, and $l_{(i,j)}$ is the label of the edge connecting them.

Through DFS coding, a one-to-one mapping is built between a subscripted graph and a DFS code (a one-to-many mapping between a graph and DFS codes). When the context is clear, we treat a subscripted graph and its DFS code as the same. All the notations on subscripted graphs can also be applied to DFS codes. The graph represented by a DFS code α is written G_α.

Second, we define an order among edge sequences. Since one graph may have several DFS codes, we want to build an order among these codes and select one code to represent the graph. Because we are dealing with labeled graphs, the label information should be considered as one of the ordering factors. The labels of vertices and edges are used to break the tie when two edges have the exact same subscript, but different labels. Let the edge order relation $\prec_T$ take the first priority, the vertex label l_i take the second priority, the edge label $l_{(i,j)}$ take the third, and the vertex label l_j take the fourth to determine the order of two edges. For example, the first edge of the three DFS codes in Table 9.1 is $(0,1,X,a,X)$, $(0,1,X,a,X)$, and $(0,1,Y,b,X)$, respectively. All of them share the same subscript $(0,1)$. So relation $\prec_T$ cannot tell the difference among them. But using label information, following the order of first vertex label, edge label, and second vertex label,

Table 9.1 DFS code for Figure 9.7(b), 9.7(c), and 9.7(d).

edge	γ_0	γ_1	γ_2
e_0	$(0,1,X,a,X)$	$(0,1,X,a,X)$	$(0,1,Y,b,X)$
e_1	$(1,2,X,a,Z)$	$(1,2,X,b,Y)$	$(1,2,X,a,X)$
e_2	$(2,0,Z,b,X)$	$(1,3,X,a,Z)$	$(2,3,X,b,Z)$
e_3	$(1,3,X,b,Y)$	$(3,0,Z,b,X)$	$(3,1,Z,a,X)$

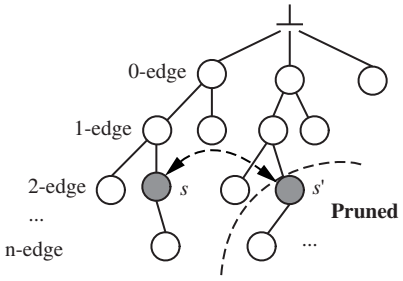

Figure 9.9 Lexicographic search tree.

we have $(0,1,X,a,X) < (0,1,Y,b,X)$. The ordering based on the above rules is called ***DFS Lexicographic Order***. According to this ordering, we have $\gamma_0 < \gamma_1 < \gamma_2$ for the DFS codes listed in Table 9.1.

Based on the DFS lexicographic ordering, the *minimum DFS code* of a given graph G, written as $\mathsf{dfs}(G)$, is the minimal one among all the DFS codes. For example, code γ_0 in Table 9.1 is the minimum DFS code of the graph in Figure 9.7(a). The subscripting that generates the minimum DFS code is called the *base subscripting*.

We have the following important relationship between the minimum DFS code and the isomorphism of the two graphs: *Given two graphs G and $G^{'}$, G is isomorphic to $G^{'}$ if and only if* $\mathsf{dfs}(G) = \mathsf{dfs}(G^{'})$. Based on this property, what we need to do for mining frequent subgraphs is to perform only the right-most extensions on the minimum DFS codes since such an extension will guarantee the completeness of mining results.

Figure 9.9 shows how to arrange all DFS codes in a search tree through right-most extensions. The root is an empty code. Each node is a DFS code encoding a graph. Each edge represents a right-most extension from a $(k-1)$-length DFS code to a k-length DSF code. The tree itself is ordered: left siblings are smaller than right siblings in the sense of DFS lexicographic order. Because any graph has at least one DFS code, the search tree can enumerate all possible subgraphs in a graph data set. However, one graph may have several DFS codes, minimum and nonminimum. The search of nonminimum DFS codes does not produce useful results. "*Is it necessary to perform right-most extension on nonminimum DFS codes?*" The answer is "*no*." If codes s and s' in Figure 9.9 encode the same graph, the search space under s' can be safely pruned.

The details of gSpan are depicted in Figure 9.10. gSpan is called recursively to extend graph patterns so that their frequent descendants are found until their support is lower than minsup or its code is not minimum any more. The difference between gSpan and PatternGrowth is at the right-most extension and extension termination of nonminimum DFS codes (lines 1-2). We replace the existence judgement in lines 1-2 of PatternGrowth with the inequation $s \neq dfs(s)$. Actually, $s \neq dfs(s)$ is more efficient to calculate. Line 5 requires exhaustive enumeration of s in D in order to count the frequency of all the possible right-most extensions of s.

The algorithm of Figure 9.10 implements a depth-first search version of gSpan. Actually, breadth-first search works too: for each newly discovered frequent subgraph in line 8,

Algorithm: gSpan. Pattern growth-based frequent substructure mining that reduces duplicate graph generation.
Input:

- s, a DFS code;
- D, a graph data set;
- *min_sup*, the minimum support threshold.

Output:

- The frequent graph set S.

Method:

(1) **if** $s \neq dfs(s)$, **then**

(2) **return**;

(3) insert s into S;

(4) set C to $\varnothing$;

(5) scan D once, find all the edges e such that s can be *right-most* extended to $s \diamond_r e$; insert $s \diamond_r e$ into C and count its frequency;

(6) sort C in DFS lexicographic order;

(7) **for each** frequent $s \diamond_r e$ in C **do**

(8) *gSpan*$(s \diamond_r e, D, min\_sup, S)$;

(9) **return**;

Figure 9.10 gSpan: A pattern-growth algorithm for frequent substructure mining.

instead of directly calling gSpan, we insert it into a global first-in-first-out queue Q, which records all subgraphs that have not been extended. We then "gSpan" each subgraph in Q one by one. The performance of a breadth-first search version of gSpan is very close to that of the depth-first search, although the latter usually consumes less memory.

9.1.2 Mining Variant and Constrained Substructure Patterns

The frequent subgraph mining discussed in the previous section handles only one special kind of graph: *labeled, undirected, connected simple graphs without any specific constraints*.

That is, we assume that the database to be mined contains a set of graphs each consisting of a set of labeled vertices and labeled but undirected edges, with no other constraints. However, many applications or users may need to enforce various kinds of *constraints* on the patterns to be mined or seek *variant substructure patterns*. For example, we may like to mine patterns, each of which contain certain specific vertices/edges, or where the total number of vertices/edges is within a specified range. Or what if we seek patterns where the average density of the graph patterns is above a threshold? Although it is possible to develop customized algorithms for each such case, there are too many variant cases to consider. Instead, a general framework is needed—one that can classify constraints on the graph patterns. Efficient constraint-based methods can then be developed for mining substructure patterns and their variants. In this section, we study several variants and constrained substructure patterns and look at how they can be mined.

Mining Closed Frequent Substructures

The first important variation of a frequent substructure is the **closed frequent substructure**. Take mining frequent subgraphs as an example. As with frequent itemset mining and sequential pattern mining, mining graph patterns may generate an explosive number of patterns. This is particularly true for dense data sets because all of the subgraphs of a frequent graph are also frequent. This is an inherent problem because according to the Apriori property, all the subgraphs of a frequent substructure must be frequent. A large graph pattern may generate an exponential number of frequent subgraphs. For example, among 423 confirmed active chemical compounds in an AIDS antiviral screen data set, there are nearly 1 million frequent graph patterns whose support is at least 5%. This renders the further analysis on frequent graphs nearly impossible.

One way to alleviate this problem is to mine only frequent closed graphs, where a frequent graph G is **closed** if and only if there is no proper supergraph G' that has the same support as G. Alternatively, we can mine maximal subgraph patterns where a frequent pattern G is **maximal** if and only if there is no frequent super-pattern of G. A set of closed subgraph patterns has the same expressive power as the full set of subgraph patterns under the same minimum support threshold, because the latter can be generated by the derived set of closed graph patterns. On the other hand, the maximal pattern set is a subset of the closed pattern set. It is usually more compact than the closed pattern set. However, we cannot use it to reconstruct the entire set of frequent patterns—the support information of a pattern is lost if it is a proper subpattern of a maximal pattern, yet carries a different support.

Example 9.3 **Maximal frequent graph.** The two graphs in Figure 9.2 are closed frequent graphs, but only the first graph is a maximal frequent graph. The second graph is not maximal because it has a frequent supergraph. ■

Mining closed graphs leads to a complete but more compact representation. For example, for the AIDS antiviral data set mentioned above, among the 1 million frequent graphs, only about 2,000 are closed frequent graphs. If further analysis, such as

classification or clustering, is performed on closed frequent graphs instead of frequent graphs, it will achieve similar accuracy with less redundancy and higher efficiency.

An efficient method, called CloseGraph, was developed for mining closed frequent graphs by extension of the gSpan algorithm. Experimental study has shown that CloseGraph often generates far fewer graph patterns and runs more efficiently than gSpan, which mines the full pattern set.

Extension of Pattern-Growth Approach: Mining Alternative Substructure Patterns

A typical pattern-growth graph mining algorithm, such as gSpan or CloseGraph, mines *labeled, connected, undirected* frequent or closed subgraph patterns. Such a graph mining framework can easily be extended for mining *alternative substructure patterns*. Here we discuss a few such alternatives.

First, the method can be extended for **mining unlabeled or partially labeled graphs**. Each vertex and each edge in our previously discussed graphs contain labels. Alternatively, if none of the vertices and edges in a graph are labeled, the graph is **unlabeled**. A graph is **partially labeled** if only some of the edges and/or vertices are labeled. To handle such cases, we can build a label set that contains the original label set and a new empty label, ϕ. Label ϕ is assigned to vertices and edges that do not have labels. Notice that label ϕ may match with any label or with ϕ only, depending on the application semantics. With this transformation, gSpan (and CloseGraph) can directly mine unlabeled or partially labeled graphs.

Second, we examine whether gSpan can be extended to **mining nonsimple graphs**. A **nonsimple graph** may have a *self-loop* (i.e., an edge joins a vertex to itself) and *multiple edges* (i.e., several edges connecting two of the same vertices). In gSpan, we always first grow backward edges and then forward edges. In order to accommodate self-loops, the growing order should be changed to *backward edges, self-loops, and forward edges*. If we allow sharing of the same vertices in two neighboring edges in a DFS code, the definition of DFS lexicographic order can handle multiple edges smoothly. Thus gSpan can mine nonsimple graphs efficiently too.

Third, we see how gSpan can be extended to handle **mining directed graphs**. In a directed graph, each edge of the graph has a defined direction. If we use a 5-tuple, $(i, j, l_i, l_{(i,j)}, l_j)$, to represent an undirected edge, then for directed edges, a new state is introduced to form a 6-tuple, $(i, j, d, l_i, l_{(i,j)}, l_j)$, where d represents the direction of an edge. Let $d = +1$ be the direction from i (v_i) to j (v_j), whereas $d = -1$ be that from j (v_j) to i (v_i). Notice that the sign of d is not related with the forwardness or backwardness of an edge. When extending a graph with one more edge, this edge may have two choices of d, which only introduces a new state in the growing procedure and need not change the framework of gSpan.

Fourth, the method can also be extended to **mining disconnected graphs**. There are two cases to be considered: (1) the graphs in the data set may be disconnected, and (2) the graph patterns may be disconnected. For the first case, we can transform the original data set by adding a virtual vertex to connect the disconnected graphs in each

graph. We then apply gSpan on the new graph data set. For the second case, we redefine the DFS code. A disconnected graph pattern can be viewed as a set of connected graphs, $r = \{g_0, g_1, \ldots, g_m\}$, where g_i is a connected graph, $0 \leq i \leq m$. Because each graph can be mapped to a minimum DFS code, a disconnected graph r can be translated into a code, $\gamma = (s_0, s_1, \ldots, s_m)$, where s_i is the minimum DFS code of g_i. The order of g_i in r is irrelevant. Thus, we enforce an order in $\{s_i\}$ such that $s_0 \leq s_1 \leq \ldots \leq s_m$. γ can be extended by either adding one-edge s_{m+1} ($s_m \leq s_{m+1}$) or by extending s_m, ..., and s_0. When checking the frequency of γ in the graph data set, make sure that $g_0, g_1, \ldots,$ and g_m are disconnected with each other.

Finally, if we view a tree as a degenerated graph, it is straightforward to extend the method to **mining frequent subtrees**. In comparison with a general graph, a tree can be considered as a degenerated direct graph that does not contain any edges that can go back to its parent or ancestor nodes. Thus if we consider that our traversal always starts at the root (because the tree does not contain any backward edges), gSpan is ready to mine tree structures. Based on the mining efficiency of the pattern growth–based approach, it is expected that gSpan can achieve good performance in tree-structure mining.

Constraint-Based Mining of Substructure Patterns

As we have seen in previous chapters, various kinds of constraints can be associated with a user's mining request. Rather than developing many case-specific substructure mining algorithms, it is more appropriate to set up a general framework of constraint-based substructure mining so that systematic strategies can be developed to push constraints deep into the mining process.

Constraint-based mining of frequent substructures can be developed systematically, similar to the constraint-based mining of frequent patterns and sequential patterns introduced in Chapters 5 and 8. Take graph mining as an example. As with the constraint-based frequent pattern mining framework outlined in Chapter 5, graph constraints can be classified into a few categories, including *antimonotonic, monotonic*, and *succinct*. Efficient constraint-based mining methods can be developed in a similar way by extending efficient graph-pattern mining algorithms, such as gSpan and CloseGraph.

Example 9.4 **Constraint-based substructure mining.** Let's examine a few commonly encountered classes of constraints to see how the constraint-pushing technique can be integrated into the pattern-growth mining framework.

1. **Element, set, or subgraph containment constraint**. Suppose a user requires that the mined pattern contain a particular set of subgraphs. This is a **succinct constraint**, which can be pushed deep into the beginning of the mining process. That is, we can take the given set of subgraphs as a query, perform selection first using the constraint, and then mine on the selected data set by growing (i.e., extending) the patterns from the given set of subgraphs. A similar strategy can be developed if we require that the mined graph pattern must contain a particular set of edges or vertices.

2. **Geometric constraint**. A geometric constraint can be that the angle between each pair of connected edges must be within a range, written as "$C_G = min\_angle \leq angle(e_1, e_2, v, v_1, v_2) \leq max\_angle$," where two edges e_1 and e_2 are connected at vertex v with the two vertices at the other ends as v_1 and v_2, respectively. C_G is an **antimonotonic constraint** because if one angle in a graph formed by two edges does not satisfy C_G, further growth on the graph will never satisfy C_G. Thus C_G can be pushed deep into the edge growth process and reject any growth that does not satisfy C_G.
3. **Value-sum constraint**. For example, such a constraint can be that the sum of (positive) weights on the edges, Sum_e, be within a range *low* and *high*. This constraint can be split into two constraints, $Sum_e \geq low$ and $Sum_e \leq high$. The former is a **monotonic constraint**, because once it is satisfied, further "growth" on the graph by adding more edges will always satisfy the constraint. The latter is an **antimonotonic constraint**, because once the condition is not satisfied, further growth of Sum_e will never satisfy it. The constraint pushing strategy can then be easily worked out. ■

Notice that a graph-mining query may contain multiple constraints. For example, we may want to mine graph patterns satisfying constraints on both the geometric and minimal sum of edge weights. In such cases, we should try to push multiple constraints simultaneously, exploring a method similar to that developed for frequent itemset mining. For the multiple constraints that are difficult to push in simultaneously, customized constraint-based mining algorithms should be developed accordingly.

Mining Approximate Frequent Substructures

An alternative way to reduce the number of patterns to be generated is to mine approximate frequent substructures, which allow slight structural variations. With this technique, we can represent several slightly different frequent substructures using one approximate substructure.

The principle of *minimum description length* (Chapter 6) is adopted in a substructure discovery system called SUBDUE, which mines approximate frequent substructures. It looks for a substructure pattern that can best compress a graph set based on the Minimum Description Length (MDL) principle, which essentially states that the simplest representation is preferred. SUBDUE adopts a constrained beam search method. It grows a single vertex incrementally by expanding a node in it. At each expansion, it searches for the best total description length: the description length of the pattern and the description length of the graph set with all the instances of the pattern condensed into single nodes. SUBDUE performs approximate matching to allow slight variations of substructures, thus supporting the discovery of approximate substructures.

There should be many different ways to mine approximate substructure patterns. Some may lead to a better representation of the entire set of substructure patterns, whereas others may lead to more efficient mining techniques. More research is needed in this direction.

Mining Coherent Substructures

A frequent substructure G is a **coherent subgraph** if the mutual information between G and each of its own subgraphs is above some threshold. The number of coherent substructures is significantly smaller than that of frequent substructures. Thus, mining coherent substructures can efficiently prune redundant patterns (i.e., patterns that are similar to each other and have similar support). A promising method was developed for mining such substructures. Its experiments demonstrate that in mining spatial motifs from protein structure graphs, the discovered coherent substructures are usually statistically significant. This indicates that coherent substructure mining selects a small subset of features that have high distinguishing power between protein classes.

Mining Dense Substructures

In the analysis of graph pattern mining, researchers have found that there exists a specific kind of graph structure, called a **relational graph**, where each node label is used only once per graph. The relational graph is widely used in modeling and analyzing massive networks (e.g., biological networks, social networks, transportation networks, and the World Wide Web). In biological networks, nodes represent objects like genes, proteins, and enzymes, whereas edges encode the relationships, such as control, reaction, and correlation between these objects. In social networks, each node represents a unique entity, and an edge describes a kind of relationship between entities. One particular interesting pattern is the **frequent highly connected** or **dense** subgraph in large relational graphs. In social networks, this kind of pattern can help identify groups where people are strongly associated. In computational biology, a highly connected subgraph could represent a set of genes within the same functional module (i.e., a set of genes participating in the same biological pathways).

This may seem like a simple constraint-pushing problem using the minimal or average degree of a vertex, where the **degree** of a vertex v is the number of edges that connect v. Unfortunately, things are not so simple. Although average degree and minimum degree display some level of connectivity in a graph, they cannot guarantee that the graph is connected in a balanced way. Figure 9.11 shows an example where some part of a graph may be loosely connected even if its average degree and minimum degree are both high. The removal of edge e_1 would make the whole graph fall apart. We may enforce the

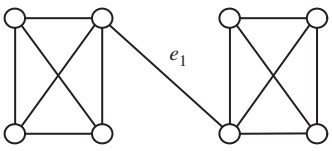

Figure 9.11 Average Degree: 3.25, Minimum Degree: 3.

following downward closure constraint: a graph is highly connected if and only if each of its connected subgraphs is highly connected. However, some global tightly connected graphs may not be locally well connected. It is too strict to have this downward closure constraint. Thus, we adopt the concept of *edge connectivity*, as follows: Given a graph G, an **edge cut** is a set of edges E_c such that $E(G) - E_c$ is disconnected. A **minimum cut** is the smallest set in all edge cuts. The **edge connectivity** of G is the size of a minimum cut. A graph is **dense** if its edge connectivity is no less than a specified minimum cut threshold.

Now the problem becomes how to mine closed frequent dense relational graphs that satisfy a user-specified connectivity constraint. There are two approaches to mining such closed dense graphs efficiently: a pattern-growth approach called CloseCut and a pattern-reduction approach, called Splat. We briefly outline their ideas as follows.

Similar to pattern-growth frequent itemset mining, **CloseCut** first starts with a small frequent candidate graph, extends it as much as possible by adding new edges until it finds the largest supergraph with the same support (i.e., its closed supergraph). The discovered graph is decomposed to extract subgraphs satisfying the connectivity constraint. It then extends the candidate graph by adding new edges, and repeats the above operations until no candidate graph is frequent.

Instead of enumerating graphs from small ones to large ones, **Splat** directly intersects relational graphs to obtain highly connected graphs. Let pattern g be a highly connected graph in relational graphs $G_{i_1}, G_{i_2}, \ldots$, and G_{i_l} $(i_1 < i_2 < \ldots < i_l)$. In order to mine patterns in a larger set $\{G_{i_1}, G_{i_2}, \ldots, G_{i_l}, G_{i_{l+1}}\}$, Splat intersects g with graph $G_{i_{l+1}}$. Let $g' = g \cap G_{i_{l+1}}$. Some edges in g may be removed because they do not exist in graph $G_{i_{l+1}}$. Thus, the connectivity of the new graph g' may no longer satisfy the constraint. If so, g' is decomposed into smaller highly connected subgraphs. We progressively reduce the size of candidate graphs by intersection and decomposition operations. We call this approach a **pattern-reduction approach**.

Both methods have shown good scalability in large graph data sets. CloseCut has better performance on patterns with high support and low connectivity. On the contrary, Splat can filter frequent graphs with low connectivity in the early stage of mining, thus achieving better performance for the high-connectivity constraints. Both methods are successfully used to extract interesting patterns from multiple biological networks.

9.1.3 Applications: Graph Indexing, Similarity Search, Classification, and Clustering

In the previous two sections, we discussed methods for mining various kinds of frequent substructures. There are many interesting applications of the discovered structured patterns. These include building graph indices in large graph databases, performing similarity search in such data sets, characterizing structure data sets, and classifying and clustering the complex structures. We examine such applications in this section.

- CrossMine and CrossClus are methods for **multirelational classification** and **multirelational clustering**, respectively. Both use tuple ID propagation to avoid physical joins. In addition, CrossClus employs user guidance to constrain the search space.

Exercises

9.1 Given two predefined sets of graphs, *contrast patterns* are substructures that are frequent in one set but infrequent in the other. Discuss how to mine contrast patterns efficiently in large graph data sets.

9.2 Multidimensional information can be associated with the vertices and edges of each graph. Study how to develop efficient methods for mining *multidimensional graph patterns*.

9.3 *Constraints* often play an important role in efficient graph mining. There are many potential constraints based on users' requests in graph mining. For example, one may want graph patterns containing or excluding certain vertices (or edges), with minimal or maximal size, containing certain subgraphs, with certain summation values, and so on. Based on how a constraint behaves in graph mining, give a systematic classification of constraints and work out rules on how to maximally use such constraints in efficient graph mining.

9.4 Our discussion of frequent graph pattern mining was confined to graph transactions (i.e., considering each graph in a graph database as a single "transaction" in a transactional database). In many applications, one needs to mine frequent subgraphs in a *large single graph* (such as the Web or a large social network). Study how to develop efficient methods for mining frequent and closed graph patterns in such data sets.

9.5 What are the challenges for *classification in a large social network* in comparison with classification in a single data relation? Suppose each node in a network represents a paper, associated with certain properties, such as author, research topic, and so on and each directed edge from node A to node B indicates that paper A cites paper B. Design an effective classification scheme that may effectively build a model for highly regarded papers on a particular topic.

9.6 A group of students are linked to each other in a social network via advisors, courses, research groups, and friendship relationships. Present a *clustering* method that may partition students into different groups according to their research interests.

9.7 Many diseases spread via people's physical contacts in public places such as offices, classrooms, buses, shopping centers, hotels, and restaurants. Suppose a database registers the concrete movement of many people (e.g., location, time, duration, and activity). Design a method that can be used to rank the "not visited" places during a virus-spreading season.

9.8 Design an effective method that discovers *hierarchical clusters in a social network*, such as a hierarchical network of friends.

9.9 Social networks evolve with time. Suppose the history of a social network is kept. Design a method that may discover the *trend of evolution* of the network.

9.10 There often exist *multiple social networks* linking a group of objects. For example, a student could be in a class, a research project group, a family member, member of a neighborhood, and so on. It is often beneficial to consider their joint effects or interactions. Design an efficient method in social network analysis that may incorporate multiple social networks in data mining.

9.11 Outline an efficient method that may find strong *correlation rules* in a large, multirelational database.

9.12 It is important to take a user's advice to cluster objects across multiple relations because many features among these relations could be relevant to the objects. A user may select a sample set of objects and claim that some should be in the same cluster but some cannot. Outline an effective clustering method with such *user guidance*.

9.13 As a result of the close relationships among multiple departments or enterprises, it is necessary to perform data mining across multiple but interlinked databases. In comparison with multirelational data mining, one major difficulty with mining across multiple databases is *semantic heterogeneity* across databases. For example, the same person "William Nelson" in one database could be "Bill Nelson" or "B. Nelson" in another one. Design a data mining method that may consolidate such objects by exploring object linkages among multiple databases.

9.14 Outline an effective method that performs *classification* across multiple heterogeneous databases.

Bibliographic Notes

Research into graph mining has developed many frequent subgraph mining methods. Washio and Motoda [WM03] performed a survey on graph-based data mining. Many well-known pairwise isomorphism testing algorithms were developed, such as Ullmann's Backtracking [Ull76] and McKay's Nauty [McK81]. Dehaspe, Toivonen, and King [DTK98] applied inductive logic programming to predict chemical carcinogenicity by mining frequent substructures. Several Apriori-based frequent substructure mining algorithms have been proposed, including AGM by Inokuchi, Washio, and Motoda [IWM98], FSG by Kuramochi and Karypis [KK01], and an edge-disjoint path-join algorithm by Vanetik, Gudes, and Shimony [VGS02]. Pattern-growth-based graph pattern mining algorithms include gSpan by Yan and Han [YH02], MoFa by Borgelt and Berthold [BB02], FFSM and SPIN by Huan, Wang, and Prins [HWP03] and Prins, Yang, Huan, and Wang [PYHW04], respectively, and Gaston by Nijssen and Kok [NK04]. These algorithms were inspired by PrefixSpan [PHMA$^+$01] for mining sequences, and TreeMinerV [Zak02] and FREQT [AAK$^+$02] for mining trees. A disk-based frequent graph mining method was proposed by Wang, Wang, Pei et al., [WWP$^+$04].

Mining closed graph patterns was studied by Yan and Han [YH03], with the proposal of the algorithm, CloseGraph, as an extension of gSpan and CloSpan [YHA03]. Holder, Cook, and Djoko [HCD9] proposed SUBDUE for approximate substructure

pattern discovery based on minimum description length and background knowledge. Mining coherent subgraphs was studied by Huan, Wang, Bandyopadhyay et al. [HWB+04]. For mining relational graphs, Yan, Zhou, and Han [YZH05] proposed two algorithms, CloseCut and Splat, to discover exact dense frequent substructures in a set of relational graphs.

Many studies have explored the applications of mined graph patterns. Path-based graph indexing approaches are used in GraphGrep, developed by Shasha, Wang, and Giugno [SWG02], and in Daylight, developed by James, Weininger, and Delany [JWD03]. Frequent graph patterns were used as graph indexing features in the gIndex and Grafil methods proposed by Yan, Yu, and Han [YYH04, YYH05] to perform fast graph search and structure similarity search. Borgelt and Berthold [BB02] illustrated the discovery of active chemical structures in an HIV-screening data set by contrasting the support of frequent graphs between different classes. Deshpande, Kuramochi, and Karypis [DKK02] used frequent structures as features to classify chemical compounds. Huan, Wang, Bandyopadhyay et al. [HWB+04] successfully applied the frequent graph mining technique to study protein structural families. Koyuturk, Grama, and Szpankowski [KGS04] proposed a method to detect frequent subgraphs in biological networks. Hu, Yan, Yu et al. [HYY+05] developed an algorithm called CoDense to find dense subgraphs across multiple biological networks.

There has been a great deal of research on social networks. For texts on social network analysis, see Wasserman and Faust [WF94], Degenne and Forse [DF99], Scott [Sco05], Watts [Wat03a], Barabási [Bar03], and Carrington, Scott, and Wasserman [CSW05]. For a survey of work on social network analysis, see Newman [New03]. Barabási, Oltvai, Jeong et al. have several comprehensive tutorials on the topic, available at *http://www.nd.edu/∼networks/publications.htm#talks0001*. Books on small world networks include Watts [Wat03b] and Buchanan [Buc03]. Milgram's "six degrees of separation" experiment is presented in [Mil67].

The *Forest Fire model* for network generation was proposed in Leskovec, Kleinberg, and Faloutsos [LKF05]. The *preferential attachment model* was studied in Albert and Barbasi [AB99] and Cooper and Frieze [CF03]. The *copying model* was explored in Kleinberg, Kumar, Raghavan, et al. [KKR+99] and Kumar, Raghavan, Rajagopalan, et al., [KRR+00].

Link mining tasks and challenges were overviewed by Getoor [Get03]. A link-based classification method was proposed in Lu and Getoor [LG03]. Iterative classification and inference algorithms have been proposed for hypertext classification by Chakrabarti, Dom, and Indyk [CDI98] and Oh, Myaeng, and Lee [OML00]. Bhattacharya and Getoor [BG04] propose a method for clustering linked data, which can be used to solve the data mining tasks of entity deduplication and group discovery. A method for group discovery was proposed by Kubica, Moore, and Schneider [KMS03]. Approaches to link prediction, based on measures for analyzing the "proximity" of nodes in a network, are described in Liben-Nowell and Kleinberg [LNK03]. The Katz measure was presented in Katz [Kat53]. A probabilistic model for learning link structure is given in Getoor, Friedman, Koller, and Taskar [GFKT01]. Link prediction for counterterrorism was proposed by Krebs [Kre02]. Viral marketing was described by Domingos [Dom05] and his work

with Richardson [DR01, RD02]. BLOG (Bayesian LOGic), a language for reasoning with unknown objects, was proposed by Milch, Marthi, Russell et al. [MMR05] to address the closed world assumption problem. Mining newsgroups to partition discussion participants into opposite camps using quotation networks was proposed by Agrawal, Rajagopalan, Srikant, and Xu [ARSX04]. The relation selection and extraction approach to community mining from multirelational networks was described in Cai, Shao, He, et al. [CSH+05].

Multirelational data mining has been investigated extensively in the Inductive Logic Programming (ILP) community. Lavrac and Dzeroski [LD94] and Muggleton [Mug95] provide comprehensive introductions to Inductive Logic Programming (ILP). An overview of multirelational data mining was given by Dzeroski [Dze03]. Well-known ILP systems include FOIL by Quinlan and Cameron-Jones [QCJ93], Golem by Muggleton and Feng [MF90], and Progol by Muggleton [Mug95]. More recent systems include TILDE by Blockeel, De Raedt, and Ramon [BRR98], Mr-SMOTI by Appice, Ceci, and Malerba [ACM03], and RPTs by Neville, Jensen, Friedland, and Hay [NJFH03], which inductively constructs decision trees from relational data. Probabilistic approaches to multirelational classification include probabilistic relational models by Getoor, Friedman, Koller, and Taskar [GFKT01] and by Taskar, Segal, and Koller [TSK01]. Popescul, Ungar, Lawrence, and Pennock [PULP02] propose an approach to integrate ILP and statistical modeling for document classification and retrieval. The CrossMine approach is described in Yin, Han, Yang, and Yu [YHYY04]. The look-one-ahead method used in CrossMine was developed by Blockeel, De Raedt, and Ramon [BRR98]. Multirelational clustering was explored by Gartner, Lloyd, and Flach [GLF04], and Kirsten and Wrobel [KW98, KW00]. CrossClus performs multirelational clustering with user guidance and was proposed by Yin, Han, and Yu [YHY05].