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 look 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.

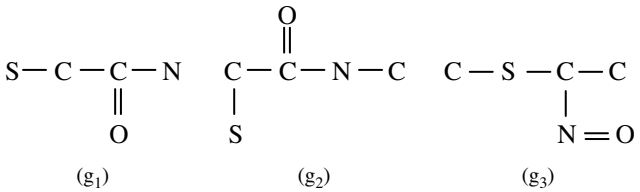

Figure 9.1 A sample graph data set.

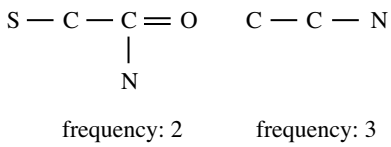

Figure 9.2 Frequent graphs.

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:

$S_1 \leftarrow$ frequent single-elements in the data set;
Call AprioriGraph(D, *min_sup*, S_1);

procedure AprioriGraph(D, *min_sup*, S_k)

(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).

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

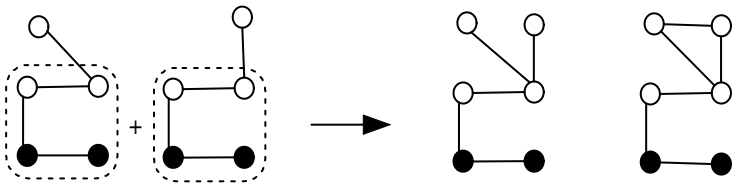

Figure 9.4 AGM: Two substructures joined by two chains.

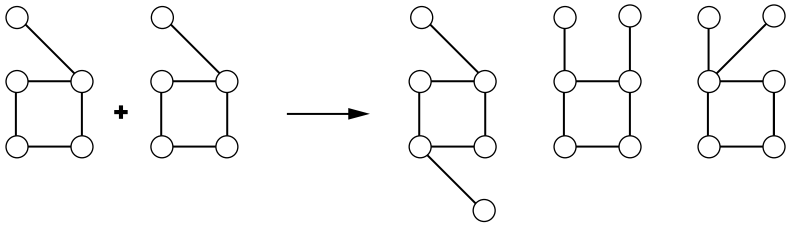

Figure 9.5 FSG: Two substructure patterns and their potential candidates.

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 the 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:

$S \leftarrow \varnothing$;
Call PatternGrowthGraph(g, D, *min_sup*, S);

procedure PatternGrowthGraph(g, D, *min_sup*, S)

(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 a second time a **duplicate graph**. Although line 1 of PatternGrowthGraph gets rid of duplicate graphs, the generation and detection of duplicate graphs may increase the workload. In order to reduce the

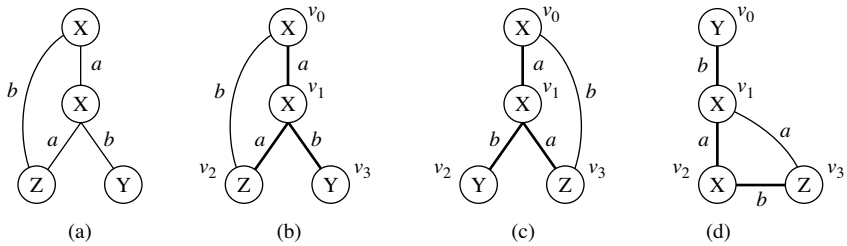

Figure 9.7 DFS subscripting.

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 a 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 another vertex on the right-most path (*backward extension*); or it can introduce a new vertex and connect to a vertex 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 into a sequence; and (2) *sequence order*, which builds an order among edge sequences (i.e., graphs).

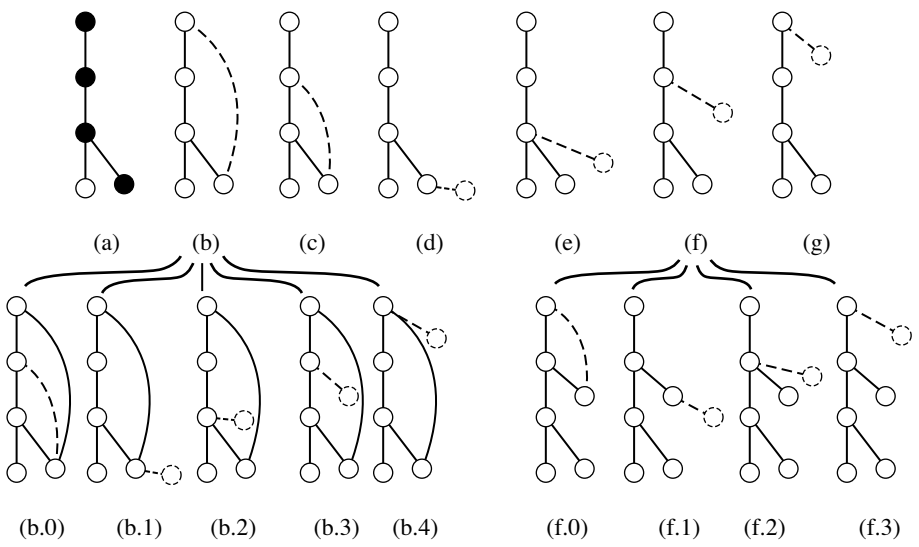

Figure 9.8 Right-most extension.

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, we have $(0,1,X,a,X) < (0,1,Y,b,X)$. The ordering based on the above rules is called

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	$(1,3,X,b,Y)$	$(3,0,Z,b,X)$	$(3,1,Z,a,X)$

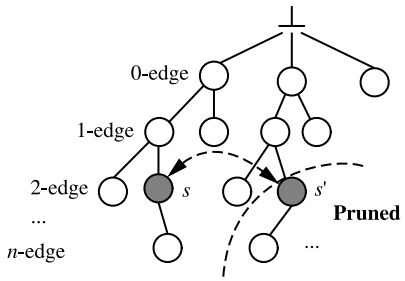

Figure 9.9 Lexicographic search tree.

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 min_sup 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

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:
$S \leftarrow \varnothing$;
Call gSpan(s, D, *min_sup*, S);

procedure PatternGrowthGraph(s, D, *min_sup*, S)

(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.

in line 8, 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 graphs: *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 contains 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 $d = -1$ is that from j (v_j) to i (v_i). Notice that the sign of d is not related to 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 patterns 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 that satisfy 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

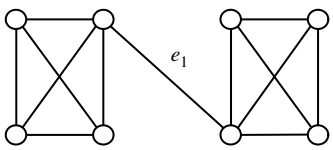

Figure 9.11 Average Degree: 3.25, Minimum Degree: 3.

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 and 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.

Graph Indexing with Discriminative Frequent Substructures

Indexing is essential for efficient search and query processing in database and information systems. Technology has evolved from single-dimensional to multidimensional indexing, claiming a broad spectrum of successful applications, including relational database systems and spatiotemporal, time-series, multimedia, text-, and Web-based information systems. However, the traditional indexing approach encounters challenges in databases involving complex objects, like graphs, because a graph may contain an exponential number of subgraphs. It is ineffective to build an index based on vertices or edges, because such features are nonselective and unable to distinguish graphs. On the other hand, building index structures based on subgraphs may lead to an explosive number of index entries.

Recent studies on graph indexing have proposed a *path-based indexing* approach, which takes the *path* as the basic indexing unit. This approach is used in the GraphGrep and Daylight systems. The general idea is to enumerate all of the existing paths in a database up to *maxL* length and index them, where a **path** is a vertex sequence, $v_1, v_2, \ldots, v_k$, such that, $\forall 1 \leq i \leq k-1$, (v_i, v_{i+1}) is an edge. The method uses the index to identify every graph, g_i, that contains all of the paths (up to *maxL* length) in query q. Even though paths are easier to manipulate than trees and graphs, and the index space is predefined, the path-based approach may not be suitable for complex graph queries, because the set of paths in a graph database is usually huge. The structural information in the graph is lost when a query graph is broken apart. It is likely that many false-positive answers will be returned. This can be seen from the following example.

Example 9.5 **Difficulty with the path-based indexing approach.** Figure 9.12 is a sample chemical data set extracted from an AIDS antiviral screening database.[1] For simplicity, we ignore the bond type. Figure 9.13 shows a sample query: 2,3-dimethylbutane. Assume that this query is posed to the sample database. Although only graph (c) in Figure 9.12 is the answer, graphs (a) and (b) cannot be pruned because both of them contain all of the paths existing in the query graph: c, $c-c$, $c-c-c$, and $c-c-c-c$. In this case, carbon chains (up to length 3) are not discriminative enough to distinguish the sample graphs. This indicates that the path may not be a good structure to serve as an index feature. ■

A method called gIndex was developed to build a compact and effective graph index structure. It takes frequent and discriminative substructures as index features. Frequent substructures are ideal candidates because they explore the shared structures in the data and are relatively stable to database updates. To reduce the *index size* (i.e., the number of frequent substructures that are used in the indices), the concept of **discriminative**

[1] *http://dtp.nci.nih.gov/docs/aids/aids_data.html.*

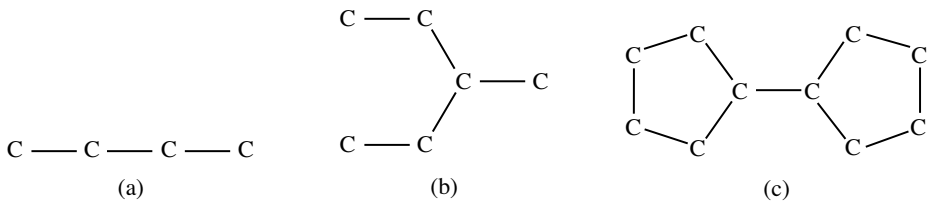

Figure 9.12 A sample chemical data set.

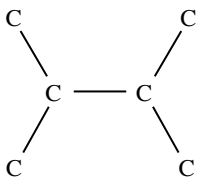

Figure 9.13 A sample query.

frequent substructure is introduced. A frequent substructure is *discriminative* if its support cannot be well approximated by the intersection of the graph sets that contain one of its subgraphs. The experiments on the AIDS antiviral data sets and others show that this method leads to far smaller index structures. In addition, it achieves similar performance in comparison with the other indexing methods, such as the path index and the index built directly on frequent substructures.

Substructure Similarity Search in Graph Databases

Bioinformatics and chem-informatics applications involve query-based search in massive, complex structural data. Even with a graph index, such search can encounter challenges because it is often too restrictive to search for an exact match of an index entry. Efficient *similarity search of complex structures* becomes a vital operation. Let's examine a simple example.

Example 9.6 **Similarity search of chemical structures.** Figure 9.14 is a chemical data set with three molecules. Figure 9.15 shows a substructure query. Obviously, no match exists for this query graph. If we relax the query by taking out one edge, then caffeine and thesal in Figure 9.14(a) and 9.14(b) will be good matches. If we relax the query further, the structure in Figure 9.14(c) could also be an answer. ■

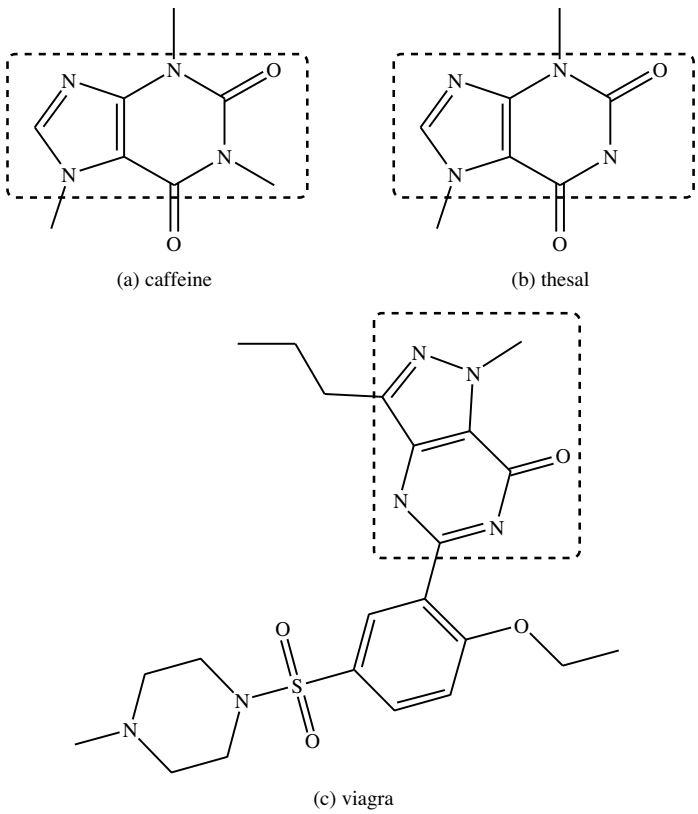

Figure 9.14 A chemical database.

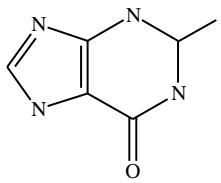

Figure 9.15 A query graph.

A naïve solution to this similarity search problem is to form a set of subgraph queries with one or more edge deletions and then use the exact substructure search. This does not work when the number of deletions is more than 1. For example, if we allow three edges to be deleted in a 20-edge query graph, it may generate $\binom{20}{3} = 1140$ substructure queries, which are too expensive to check.

A feature-based structural filtering algorithm, called Grafil (**Gra**ph Similarity **Fil**tering), was developed to filter graphs efficiently in large-scale graph databases. Grafil models each query graph as a set of features and transforms the edge deletions into "feature misses" in the query graph. It is shown that using too many features will not leverage the filtering performance. Therefore, a multifilter composition strategy is developed, where each filter uses a distinct and complementary subset of the features. The filters are constructed by a hierarchical, one-dimensional clustering algorithm that groups features with similar selectivity into a *feature set*. Experiments have shown that the multifilter strategy can significantly improve performance for a moderate relaxation ratio.

Classification and Cluster Analysis Using Graph Patterns

"*Can we apply the discovered patterns to classification and cluster analysis? If so, how?*" The discovered frequent graph patterns and/or their variants can be used as features for graph classification. First, we mine frequent graph patterns in the training set. The features that are frequent in one class but rather infrequent in the other class(es) should be considered as highly discriminative features. Such features will then be used for model construction. To achieve high-quality classification, we can adjust the thresholds on frequency, discriminativeness, and graph connectivity based on the data, the number and quality of the features generated, and the classification accuracy. Various classification approaches, including support vector machines, naïve Bayesian, and associative classification, can be used in such graph-based classification.

Similarly, cluster analysis can be explored with mined graph patterns. The set of graphs that share a large set of similar graph patterns should be considered as highly similar and should be grouped into similar clusters. Notice that the concept of graph connectivity (or minimal cuts) introduced in the section for mining frequent dense graphs can be used as an important measure to group similar graphs into clusters. In addition, the minimal support threshold can be used as a way to adjust the number of frequent clusters or generate hierarchical clusters. The graphs that do not belong to any cluster or that are far away from the derived clusters can be considered as *outliers*. Thus outliers can be considered as a by-product of cluster analysis.

Many different kinds of graphs can be discovered in graph pattern mining, especially when we consider the possibilities of setting different kinds of thresholds. Different graph patterns may likely lead to different classification and clustering results, thus it is important to consider mining graph patterns and graph classification/clustering as an intertwined process rather than a two-step process. That is, the qualities of graph classification and clustering can be improved by exploring alternative methods and thresholds when mining graph patterns.

9.2 Social Network Analysis

The notion of social networks, where relationships between entities are represented as *links* in a graph, has attracted increasing attention in the past decades. Thus social

network analysis, from a data mining perspective, is also called *link analysis* or *link mining*. In this section, we introduce the concept of social networks in Section 9.2.1, and study the characteristics of social networks in Section 9.2.2. In Section 9.2.3, we look at the tasks and challenges involved in link mining, and finally, explore exemplar forms of mining on social networks in Section 9.2.4.

9.2.1 What Is a Social Network?

From the point of view of data mining, a **social network** is a *heterogeneous* and *multirelational* data set represented by a graph. The graph is typically very large, with **nodes** corresponding to *objects* and **edges** corresponding to *links* representing relationships or interactions between objects. Both nodes and links have *attributes*. Objects may have class labels. Links can be one-directional and are not required to be binary.

Social networks need not be social in context. There are many real-world instances of technological, business, economic, and biologic social networks. Examples include electrical power grids, telephone call graphs, the spread of computer viruses, the World Wide Web, and coauthorship and citation networks of scientists. Customer networks and collaborative filtering problems (where product recommendations are made based on the preferences of other customers) are other examples. In biology, examples range from epidemiological networks, cellular and metabolic networks, and food webs, to the neural network of the nematode worm *Caenorhabditis elegans* (the only creature whose neural network has been completely mapped). The exchange of e-mail messages within corporations, newsgroups, chat rooms, friendships, sex webs (linking sexual partners), and the quintessential "old-boy" network (i.e., the overlapping boards of directors of the largest companies in the United States) are examples from sociology.

Small world (social) networks have received considerable attention as of late. They reflect the concept of "small worlds," which originally focused on networks among individuals. The phrase captures the initial surprise between two strangers ("What a small world!") when they realize that they are indirectly linked to one another through mutual acquaintances. In 1967, Harvard sociologist, Stanley Milgram, and his colleagues conducted experiments in which people in Kansas and Nebraska were asked to direct letters to strangers in Boston by forwarding them to friends who they thought might know the strangers in Boston. Half of the letters were successfully delivered through no more than five intermediaries. Additional studies by Milgram and others, conducted between other cities, have shown that there appears to be a universal "six degrees of separation" between any two individuals in the world. Examples of small world networks are shown in Figure 9.16. **Small world networks** have been characterized as having a high degree of local clustering for a small fraction of the nodes (i.e., these nodes are interconnected with one another), which at the same time are no more than a few degrees of separation from the remaining nodes. It is believed that many social, physical, human-designed, and biological networks exhibit such small world characteristics. These characteristics are further described and modeled in Section 9.2.2.

"*Why all this interest in small world networks and social networks, in general? What is the interest in characterizing networks and in mining them to learn more about their structure?*"

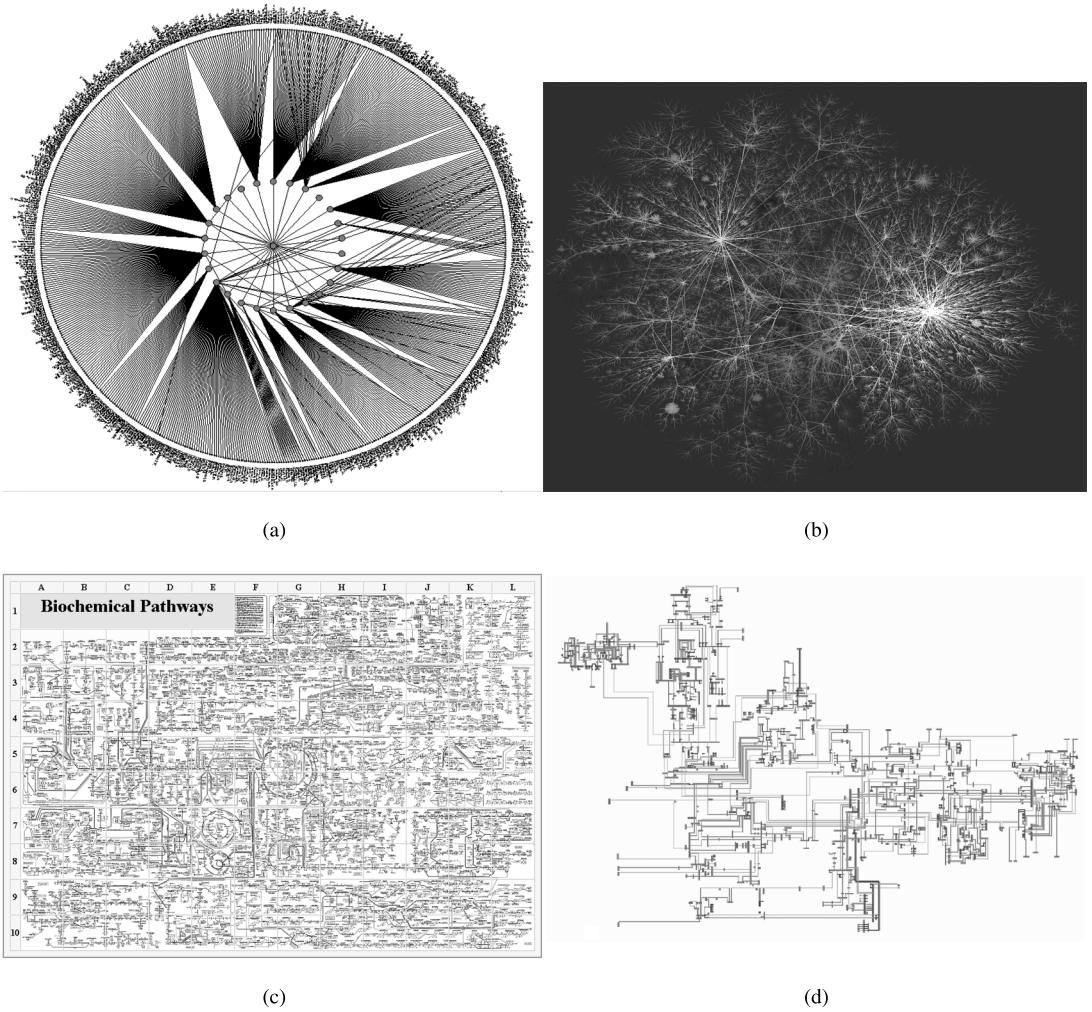

Figure 9.16 Real-world examples of social networks: (a) science coauthor network, (b) connected pages on a part of the Internet, (c) biochemical pathway network, and (d) New York state electric power grid. Figure 9.16 (a), (b), and (c) are from *www.nd.edu/∼networks/publications.html#talks0001* by Barabási, Oltvai, Jeong et al. Figure 9.11(d) is from [Str01], available at *http://tam.cornell.edu/Strogatz.html#pub*.

The reason is that structure always affects function. For example, the topology of social networks affects the spread of infectious disease through a structured population. The topology of a power grid affects the stability and robustness of its power transmission. For instance, a power failure in Cleveland, Ohio, on August 14, 2003, triggered, through an interconnecting grid system, the shutting down of nuclear power plants in New York

state and Ohio, and led to widespread power blackouts in many parts of the Northeastern United States and Southeastern Canada, which affected approximately 50 million people. The interest in networks is part of broader research in the accurate and complete description of *complex systems*. Previously, the networks available for experimental study were small and few, with very little information available regarding individual nodes. Thanks to the Internet, huge amounts of data on very large social networks are now available. These typically contain from tens of thousands to millions of nodes. Often, a great deal of information is available at the level of individual nodes. The availability of powerful computers has made it possible to probe the structure of networks. Searching social networks can help us better understand how we can reach other people. In addition, research on small worlds, with their relatively small separation between nodes, can help us design networks that facilitate the efficient transmission of information or other resources without having to overload the network with too many redundant connections. For example, it may help us design smarter search agents on the Web, which can find relevant websites in response to a query, all within the smallest number of degrees of separation from the initial website (which is, typically, a search engine).

9.2.2 Characteristics of Social Networks

As seen in the previous section, knowing the characteristics of small world networks is useful in many situations. We can build graph generation models, which incorporate the characteristics. These may be used to predict how a network may look in the future, answering "what-if" questions. Taking the Internet as an example, we may ask "*What will the Internet look like when the number of nodes doubles?*" and "*What will the number of edges be?*". If a hypothesis contradicts the generally accepted characteristics, this raises a flag as to the questionable plausibility of the hypothesis. This can help detect abnormalities in existing graphs, which may indicate fraud, spam, or Distributed Denial of Service (DDoS) attacks. Models of graph generation can also be used for simulations when real graphs are excessively large and thus, impossible to collect (such as a very large network of friendships). In this section, we study the basic characteristics of social networks as well as a model for graph generation.

"*What qualities can we look at when characterizing social networks?*" Most studies examine the **nodes' degrees**, that is, the number of edges incident to each node, and the *distances* between a pair of nodes, as measured by the *shortest path length*. (This measure embodies the small world notion that individuals are linked via short chains.) In particular, the **network diameter** is the maximum distance between pairs of nodes. Other node-to-node distances include the **average distance** between pairs and the **effective diameter** (i.e., the minimum distance, d, such that for at least 90% of the reachable node pairs, the path length is at most d).

Social networks are rarely static. Their graph representations evolve as nodes and edges are added or deleted over time. In general, social networks tend to exhibit the following phenomena:

1. **Densification power law**: Previously, it was believed that as a network evolves, the number of degrees grows linearly in the number of nodes. This was known as the

constant average degree assumption. However, extensive experiments have shown that, on the contrary, networks become increasingly *dense* over time with the average degree increasing (and hence, the number of edges growing superlinearly in the number of nodes). The densification follows the **densification power law** (or **growth power law**), which states

$$e(t) \propto n(t)^a, \tag{9.1}$$

where $e(t)$ and $n(t)$, respectively, represent the number of edges and nodes of the graph at time t, and the exponent a generally lies strictly between 1 and 2. Note that if $a = 1$, this corresponds to constant average degree over time, whereas $a = 2$ corresponds to an extremely dense graph where each node has edges to a constant fraction of all nodes.

2. **Shrinking diameter**: It has been experimentally shown that the effective diameter tends to *decrease* as the network grows. This contradicts an earlier belief that the diameter slowly increases as a function of network size. As an intuitive example, consider a citation network, where nodes are papers and a citation from one paper to another is indicated by a directed edge. The out-links of a node, v (representing the papers cited by v), are "frozen" at the moment it joins the graph. The decreasing distances between pairs of nodes consequently appears to be the result of subsequent papers acting as "bridges" by citing earlier papers from other areas.
3. **Heavy-tailed out-degree and in-degree distributions:** The number of out-degrees for a node tends to follow a heavy-tailed distribution by observing the **power law**, $1/n^a$, where n is the rank of the node in the order of decreasing out-degrees and typically, $0 < a < 2$ (Figure 9.17). The smaller the value of a, the heavier the tail. This phenomena is represented in the **preferential attachment model**, where each new node attaches to an existing network by a constant number of out-links, following a

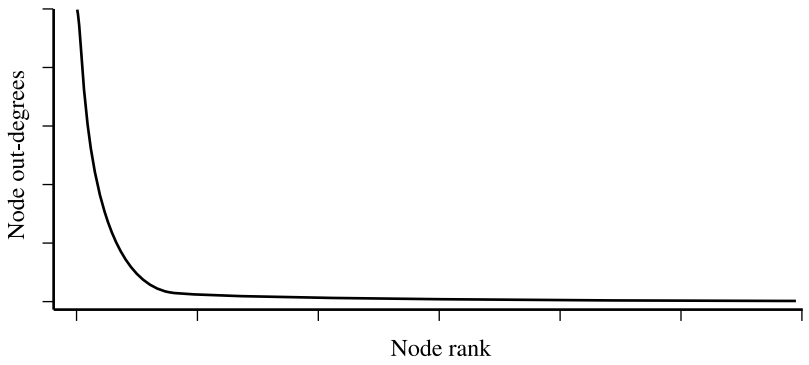

Figure 9.17 The number of out-degrees (y-axis) for a node tends to follow a heavy-tailed distribution. The node rank (x-axis) is defined as the order of deceasing out-degrees of the node.

"rich-get-richer" rule. The in-degrees also follow a heavy-tailed distribution, although it tends be more skewed than the out-degrees distribution.

A **Forest Fire model** for graph generation was proposed, which captures these characteristics of graph evolution over time. It is based on the notion that new nodes attach to the network by "burning" through existing edges in epidemic fashion. It uses two parameters, *forward burning probability*, p, and *backward burning ratio*, r, which are described below. Suppose a new node, v, arrives at time t. It attaches to G_t, the graph constructed so far, in the following steps:

1. It chooses an *ambassador node*, w, at random, and forms a link to w.
2. It selects x links incident to w, where x is a random number that is binomially distributed with mean $(1-p)^{-1}$. It chooses from out-links and in-links of w but selects in-links with probability r times lower than out-links. Let $w_1, w_2, \ldots, w_x$ denote the nodes at the other end of the selected edges.
3. Our new node, v, forms out-links to $w_1, w_2, \ldots, w_x$ and then applies step 2 recursively to each of $w_1, w_2, \ldots, w_x$. Nodes cannot be visited a second time so as to prevent the construction from cycling. The process continues until it dies out.

For an intuitive feel of the model, we return to our example of a citation network. The author of a new paper, v, initially consults w, and follows a subset of its references (which may be either forward or backward) to the papers $w_1, w_2, \ldots, w_x$. It continues accumulating references recursively by consulting these papers.

Several earlier models of network evolution were based on static graphs, identifying network characteristics from a single or small number of snapshots, with little emphasis on finding trends over time. The Forest Fire model combines the essence of several earlier models, while considering the evolution of networks over time. The *heavy-tailed out-degrees* property is observed in that, owing to the recursive nature of link formation, new nodes have a good chance of burning many edges and thus producing large out-degrees. The *heavy-tailed in-degrees* property is preserved in that Forest Fire follows the "rich-get-richer" rule: highly linked nodes can easily be reached by a new node, regardless of which ambassador the new node starts from. The flavor of a model known as the *copying model* is also observed in that a new node copies many of the neighbors of its ambassador. The *densification power law* is upheld in that a new node will have many links near the community of its ambassador, a few links beyond this, and much fewer farther away. Rigorous empirical studies found that the *shrinking diameter* property was upheld. Nodes with heavy-tailed out-degrees may serve as "bridges" that connect formerly disparate parts of the network, decreasing the network diameter.

9.2.3 Link Mining: Tasks and Challenges

"*How can we mine social networks?*" Traditional methods of machine learning and data mining, taking, as input, a random sample of homogenous objects from a single

relation, may not be appropriate here. The data comprising social networks tend to be heterogeneous, multirelational, and semi-structured. As a result, a new field of research has emerged called **link mining**. Link mining is a confluence of research in social networks, link analysis, hypertext and Web mining, graph mining, relational learning, and inductive logic programming. It embodies descriptive and predictive modeling. By considering links (the relationships between objects), more information is made available to the mining process. This brings about several new tasks. Here, we list these tasks with examples from various domains:

1. **Link-based object classification.** In traditional classification methods, objects are classified based on the attributes that describe them. Link-based classification predicts the category of an object based not only on its attributes, but also on its links, and on the attributes of linked objects.
   Web page classification is a well-recognized example of link-based classification. It predicts the category of a Web page based on word occurrence (words that occur on the page) and *anchor text* (the hyperlink words, that is, the words you click on when you click on a link), both of which serve as attributes. In addition, classification is based on links between pages and other attributes of the pages and links. In the *bibliography domain*, objects include papers, authors, institutions, journals, and conferences. A classification task is to predict the topic of a paper based on word occurrence, citations (other papers that cite the paper), and cocitations (other papers that are cited within the paper), where the citations act as links. An example from *epidemiology* is the task of predicting the disease type of a patient based on characteristics (e.g., symptoms) of the patient, and on characteristics of other people with whom the patient has been in contact. (These other people are referred to as the patients' *contacts*.)

2. **Object type prediction.** This predicts the type of an object, based on its attributes and its links, and on the attributes of objects linked to it. In the bibliographic domain, we may want to predict the venue type of a publication as either conference, journal, or workshop. In the *communication domain*, a similar task is to predict whether a communication contact is by e-mail, phone call, or mail.

3. **Link type prediction.** This predicts the type or purpose of a link, based on properties of the objects involved. Given epidemiological data, for instance, we may try to predict whether two people who know each other are family members, coworkers, or acquaintances. In another example, we may want to predict whether there is an advisor-advisee relationship between two coauthors. Given Web page data, we can try to predict whether a link on a page is an advertising link or a navigational link.

4. **Predicting link existence.** Unlike link type prediction, where we know a connection exists between two objects and we want to predict its type, instead we may want to predict whether a link exists between two objects. Examples include predicting whether there will be a link between two Web pages, and whether a paper will cite

another paper. In epidemiology, we can try to predict with whom a patient came in contact.

5. **Link cardinality estimation.** There are two forms of link cardinality estimation. First, we may predict the number of links to an object. This is useful, for instance, in predicting the authoritativeness of a Web page based on the number of links to it (in-links). Similarly, the number of out-links can be used to identify Web pages that act as *hubs*, where a hub is one or a set of Web pages that point to many authoritative pages of the same topic. In the bibliographic domain, the number of citations in a paper may indicate the impact of the paper—the more citations the paper has, the more influential it is likely to be. In epidemiology, predicting the number of links between a patient and his or her contacts is an indication of the potential for disease transmission.

   A more difficult form of link cardinality estimation predicts the number of objects reached along a path from an object. This is important in estimating the number of objects that will be returned by a query. In the Web page domain, we may predict the number of pages that would be retrieved by crawling a site (where *crawling* refers to a methodological, automated search through the Web, mainly to create a copy of all of the visited pages for later processing by a search engine). Regarding citations, we can also use link cardinality estimation to predict the number of citations of a specific author in a given journal.

6. **Object reconciliation.** In object reconciliation, the task is to predict whether two objects are, in fact, the same, based on their attributes and links. This task is common in information extraction, duplication elimination, object consolidation, and citation matching, and is also known as *record linkage* or *identity uncertainty*. Examples include predicting whether two websites are mirrors of each other, whether two citations actually refer to the same paper, and whether two apparent disease strains are really the same.

7. **Group detection.** Group detection is a clustering task. It predicts when a set of objects belong to the same group or cluster, based on their attributes as well as their link structure. An area of application is the identification of *Web communities*, where a Web community is a collection of Web pages that focus on a particular theme or topic. A similar example in the bibliographic domain is the identification of research communities.

8. **Subgraph detection.** Subgraph identification finds characteristic subgraphs within networks. This is a form of graph search and was described in Section 9.1. An example from biology is the discovery of subgraphs corresponding to protein structures. In chemistry, we can search for subgraphs representing chemical substructures.

9. **Metadata mining.** Metadata are data about data. Metadata provide semi-structured data about unstructured data, ranging from text and Web data to multimedia databases. It is useful for data integration tasks in many domains. Metadata mining can be used for *schema mapping* (where, say, the attribute *customer_id* from one database is mapped to *cust_number* from another database because they both refer to the

same entity); *schema discovery*, which generates schema from semi-structured data; and *schema reformulation*, which refines the schema based on the mined metadata. Examples include matching two bibliographic sources, discovering schema from unstructured or semi-structured data on the Web, and mapping between two medical ontologies.

In summary, the exploitation of link information between objects brings on additional tasks for link mining in comparison with traditional mining approaches. The implementation of these tasks, however, invokes many challenges. We examine several of these challenges here:

1. **Logical versus statistical dependencies.** Two types of dependencies reside in the graph—*link structures* (representing the logical relationship between objects) and *probabilistic dependencies* (representing statistical relationships, such as correlation between attributes of objects where, typically, such objects are logically related). The coherent handling of these dependencies is also a challenge for multirelational data mining, where the data to be mined exist in multiple tables. We must search over the different possible logical relationships between objects, in addition to the standard search over probabilistic dependencies between attributes. This takes a huge search space, which further complicates finding a plausible mathematical model. Methods developed in inductive logic programming may be applied here, which focus on search over logical relationships.
2. **Feature construction.** In link-based classification, we consider the attributes of an object as well as the attributes of objects linked to it. In addition, the links may also have attributes. The goal of *feature construction* is to construct a single feature representing these attributes. This can involve feature selection and feature aggregation. In *feature selection*, only the most discriminating features are included.[2] *Feature aggregation* takes a multiset of values over the set of related objects and returns a summary of it. This summary may be, for instance, the mode (most frequently occurring value); the mean value of the set (if the values are numerical); or the median or "middle" value (if the values are ordered). However, in practice, this method is not always appropriate.
3. **Instances versus classes.** This alludes to whether the model refers explicitly to individuals or to classes (generic categories) of individuals. An advantage of the former model is that it may be used to connect particular individuals with high probability. An advantage of the latter model is that it may be used to generalize to new situations, with different individuals.
4. **Collective classification and collective consolidation.** Consider training a model for classification, based on a set of class-labeled objects. Traditional classification

[2] Feature (or attribute) selection was introduced in Chapter 2.

methods consider only the attributes of the objects. After training, suppose we are given a new set of unlabeled objects. Use of the model to infer the class labels for the new objects is complicated due to possible correlations between objects—the labels of linked objects may be correlated. Classification should therefore involve an additional iterative step that updates (or consolidates) the class label of each object based on the labels of objects linked to it. In this sense, classification is done *collectively* rather than independently.

5. **Effective use of labeled and unlabeled data.** A recent strategy in learning is to incorporate a mix of both labeled and unlabeled data. Unlabeled data can help infer the object attribute distribution. Links between unlabeled (test) data allow us to use attributes of linked objects. Links between labeled (training) data and unlabeled (test) data induce dependencies that can help make more accurate inferences.

6. **Link prediction.** A challenge in link prediction is that the prior probability of a particular link between objects is typically extremely low. Approaches to link prediction have been proposed based on a number of measures for analyzing the proximity of nodes in a network. Probabilistic models have been proposed as well. For large data sets, it may be more effective to model links at a higher level.

7. **Closed versus open world assumption.** Most traditional approaches assume that we know all the potential entities in the domain. This "closed world" assumption is unrealistic in real-world applications. Work in this area includes the introduction of a language for specifying probability distributions over relational structures that involve a varying set of objects.

8. **Community mining from multirelational networks.** Typical work on social network analysis includes the discovery of groups of objects that share similar properties. This is known as *community mining*. Web page linkage is an example, where a discovered community may be a set of Web pages on a particular topic. Most algorithms for community mining assume that there is only one social network, representing a relatively homogenous relationship. In reality, there exist multiple, heterogeneous social networks, representing various relationships. A new challenge is the mining of hidden communities on such heterogeneous social networks, which is also known as *community mining on multirelational social networks*.

These challenges will continue to stimulate much research in link mining.

9.2.4 Mining on Social Networks

In this section, we explore exemplar areas of mining on social networks, namely, link prediction, mining customer networks for viral marketing, mining newsgroups using networks, and community mining from multirelational networks. Other exemplars include characteristic subgraph detection (discussed in Section 9.1) and mining link structures on the Web (addressed in Chapter 10 on text and Web mining). Pointers to

research on link-based classification and clustering are given in the bibliographic notes and exercises.

Link Prediction: What Edges Will Be Added to the Network?

Social networks are dynamic. New links appear, indicating new interactions between objects. In the **link prediction problem**, we are given a snapshot of a social network at time t and wish to *predict the edges that will be added to the network during the interval from time t to a given future time, t'*. In essence, we seek to uncover the extent to which the evolution of a social network can be modeled using features intrinsic to the model itself. As an example, consider a social network of coauthorship among scientists. Intuitively, we may predict that two scientists who are "close" in the network may be likely to collaborate in the future. Hence, link prediction can be thought of as a contribution to the study of social network evolution models.

Approaches to link prediction have been proposed based on several measures for analyzing the "proximity" of nodes in a network. Many measures originate from techniques in graph theory and social network analysis. The general methodology is as follows: All methods assign a connection weight, $score(\boldsymbol{X}, \boldsymbol{Y})$, to pairs of nodes, $\boldsymbol{X}$ and $\boldsymbol{Y}$, based on the given proximity measure and input graph, G. A ranked list in decreasing order of $score(\boldsymbol{X}, \boldsymbol{Y})$ is produced. This gives the predicted new links in decreasing order of confidence. The predictions can be evaluated based on real observations on experimental data sets.

The simplest approach ranks pairs, $\langle \boldsymbol{X}, \boldsymbol{Y} \rangle$, by the length of their *shortest path* in G. This embodies the small world notion that all individuals are linked through short chains. (Since the convention is to rank all pairs in order of *decreasing* score, here, $score(\boldsymbol{X}, \boldsymbol{Y})$ is defined as the negative of the shortest path length.) Several measures use neighborhood information. The simplest such measure is *common neighbors*—the greater the number of neighbors that $\boldsymbol{X}$ and $\boldsymbol{Y}$ have in common, the more likely $\boldsymbol{X}$ and $\boldsymbol{Y}$ are to form a link in the future. Intuitively, if authors $\boldsymbol{X}$ and $\boldsymbol{Y}$ have never written a paper together but have many colleagues in common, the more likely they are to collaborate in the future. Other measures are based on the *ensemble of all paths* between two nodes. The *Katz* measure, for example, computes a weighted sum over all paths between $\boldsymbol{X}$ and $\boldsymbol{Y}$, where shorter paths are assigned heavier weights. All of the measures can be used in conjunction with higher-level approaches, such as *clustering*. For instance, the link prediction method can be applied to a cleaned-up version of the graph, in which spurious edges have been removed.

In experiments conducted on bibliographic citation data sets, no one method is superior to all others. Several methods significantly outperform a random predictor, which suggests that network topology can provide useful information for link prediction. The Katz measure, and variations of it based on clustering, performed consistently well, although the accuracy of prediction is still very low. Future work on link prediction may focus on finding better ways to use network topology information, as well as to improve the efficiency of node distance calculations such as by approximation.

Mining Customer Networks for Viral Marketing

Viral marketing is an application of social network mining that explores how individuals can influence the buying behavior of others. Traditionally, companies have employed **direct marketing** (where the decision to market to a particular individual is based solely on her characteristics) or **mass marketing** (where individuals are targeted based on the population segment to which they belong). These approaches, however, neglect the influence that customers can have on the purchasing decisions of others. For example, consider a person who decides to see a particular movie and persuades a group of friends to see the same film. **Viral marketing** aims to optimize the positive word-of-mouth effect among customers. It can choose to spend more money marketing to an individual if that person has many social connections. Thus, by considering the interactions between customers, viral marketing may obtain higher profits than traditional marketing, which ignores such interactions.

The growth of the Internet over the past two decades has led to the availability of many social networks that can be mined for the purposes of viral marketing. Examples include e-mail mailing lists, UseNet groups, on-line forums, instant relay chat (IRC), instant messaging, collaborative filtering systems, and knowledge-sharing sites. **Knowledge-sharing sites** (such as Epinions at *www.epinions.com*) allow users to offer advice or rate products to help others, typically for free. Users can rate the usefulness or "trustworthiness" of a review, and may possibly rate other reviewers as well. In this way, a network of trust relationships between users (known as a "web of trust") evolves, representing a social network for mining.

The **network value** of a customer is the expected increase in sales to *others* that results from marketing to that customer. In the example given, if our customer convinces others to see a certain movie, then the movie studio is justified in spending more money on promoting the film to her. If, instead, our customer typically listens to others when deciding what movie to see, then marketing spent on her may be a waste of resources. Viral marketing considers a customer's network value. Ideally, we would like to mine a customer's network (e.g., of friends and relatives) to predict how probable she is to buy a certain product based not only on the characteristics of the customer, but also on the influence of the customer's neighbors in the network. If we market to a particular set of customers then, through viral marketing, we may query the *expected profit from the entire network*, after the influence of those customers has propagated throughout. This would allow us to search for the optimal set of customers to which to market. Considering the network value of customers (which is overlooked by traditional direct marketing), this may result in an improved marketing plan.

Given a set of n potential customers, let X_i be a Boolean variable that is set to 1 if customer i purchases the product being marketed, and 0 otherwise. The neighbors of X_i are the customers who directly influence X_i. M_i is defined as the *marketing action* that is taken for customer i. M_i could be Boolean (such as, set to 1 if the customer is sent a coupon, and 0 otherwise) or categoric (indicating which of several possible actions is taken). Alternatively, M_i may be continuous-valued (indicating the size of the discount offered, for example). We would like to find the marketing plan that maximizes profit.

A probabilistic model was proposed that optimizes M_i as a continuous value. That is, it optimizes the amount of marketing money spent on each customer, rather than just making a binary decision on whether to target the customer.

The model considers the following factors that influence a customer's network value. First, the customer should have high connectivity in the network and also give the product a good rating. If a highly-connected customer gives a negative review, her network value can be negative, in which case, marketing to her is not recommended. Second, the customer should have more influence on others (preferably, much more) than they have on her. Third, the *recursive* nature of this word-of-mouth type of influence should be considered. A customer may influence acquaintances, who in turn, may like the product and influence other people, and so on, until the whole network is reached. The model also incorporates another important consideration: it may pay to lose money on some customers if they are influential enough in a positive way. For example, giving a product for free to a well-selected customer may pay off many times in sales to other customers. This is a big twist from traditional direct marketing, which will only offer a customer a discount if the expected profits from the customer alone exceed the cost of the offer. The model takes into consideration the fact that we have only partial knowledge of the network and that gathering such knowledge can have an associated cost.

The task of finding the optimal set of customers is formalized as a well-defined optimization problem: *find the set of customers that maximizes the net profits*. This problem is known to be NP-hard (intractable); however, it can be approximated within 63% of the optimal using a simple hill-climbing search procedure. Customers are added to the set as long as this improves overall profit. The method was found to be robust in the presence of incomplete knowledge of the network.

Viral marketing techniques may be applied to other areas that require a large social outcome with only limited resources. Reducing the spread of HIV, combatting teenage smoking, and grass-roots political initiative are some examples. The application of viral marketing techniques to the Web domain, and vice versa, is an area of further research.

Mining Newsgroups Using Networks

Web-based social network analysis is closely related to Web mining, a topic to be studied in the next chapter. There we will introduce two popular Web page ranking algorithms, PageRank and HITS, which are proposed based on the fact that a link of Web page A to B usually indicates the endorsement of B by A.

The situation is rather different in newsgroups on topic discussions. A typical newsgroup posting consists of one or more quoted lines from another posting followed by the opinion of the author. Such quoted responses form "quotation links" and create a network in which the vertices represent individuals and the links "responded-to" relationships. An interesting phenomenon is that people more frequently respond to a message when they *disagree* than when they *agree*. This behavior exists in many newsgroups and is in sharp contrast to the Web page link graph, where linkage is an indicator of agreement or common interest. Based on this behavior, one can effectively classify and

partition authors in the newsgroup into *opposite camps* by analyzing the graph structure of the responses.

This newsgroup classification process can be performed using a graph-theoretic approach. The *quotation network* (or *graph)* can be constructed by building a quotation link between person i and person j if i has quoted from an earlier posting written by j. We can consider any bipartition of the vertices into two sets: F represents those *for* an issue and A represents those *against* it. If most edges in a newsgroup graph represent disagreements, then the optimum choice is to maximize the number of edges across these two sets. Because it is known that theoretically the *max-cut problem* (i.e., maximizing the number of edges to cut so that a graph is partitioned into two disconnected subgraphs) is an NP-hard problem, we need to explore some alternative, practical solutions. In particular, we can exploit two additional facts that hold in our situation: (1) rather than being a general graph, our instance is largely a bipartite graph with some noise edges added, and (2) neither side of the bipartite graph is much smaller than the other. In such situations, we can transform the problem into a minimum-weight, approximately balanced cut problem, which in turn can be well approximated by computationally simple spectral methods. Moreover, to further enhance the classification accuracy, we can first manually categorize a small number of prolific posters and tag the corresponding vertices in the graph. This information can then be used to bootstrap a better overall partitioning by enforcing the constraint that those classified on one side by human effort should remain on that side during the algorithmic partitioning of the graph.

Based on these ideas, an efficient algorithm was proposed. Experiments with some newsgroup data sets on several highly debatable social topics, such as abortion, gun control, and immigration, demonstrate that links carry less noisy information than text. Methods based on linguistic and statistical analysis of text yield lower accuracy on such newsgroup data sets than that based on the link analysis shown earlier because the vocabulary used by the opponent sides tends to be largely identical, and many newsgroup postings consist of too-brief text to facilitate reliable linguistic analysis.

Community Mining from Multirelational Networks

With the growth of the Web, *community mining* has attracted increasing attention. A great deal of such work has focused on mining implicit communities of Web pages, of scientific literature from the Web, and of document citations. In principle, a **community** can be defined as a group of objects sharing some common properties. **Community mining** can be thought of as subgraph identification. For example, in Web page linkage, two Web pages (objects) are related if there is a hyperlink between them. A graph of Web page linkages can be mined to identify a community or set of Web pages on a particular topic.

Most techniques for graph mining and community mining are based on a homogenous graph, that is, they assume only one kind of relationship exists between the objects. However, in real social networks, there are always various kinds of relationships

between the objects. Each relation can be viewed as a **relation network**. In this sense, the multiple relations form a **multirelational social network** (also referred to as a **heterogeneous social network**). Each kind of relation may play a distinct role in a particular task. Here, the different relation graphs can provide us with different communities.

To find a community with certain properties, we first need to identify which relation plays an important role in such a community. Such a relation might not exist explicitly, that is, we may need to first discover such a *hidden relation* before finding the community on such a relation network. Different users may be interested in different relations within a network. Thus, if we mine networks by assuming only one kind of relation, we may end up missing out on a lot of valuable hidden community information, and such mining may not be adaptable to the diverse information needs of various users. This brings us to the problem of *multirelational community mining*, which involves the mining of hidden communities on heterogeneous social networks.

Let us consider a simple example. In a typical human community, there may exist many relations: some people work at the same place; some share the same interests; some go to the same hospital, and so on. Mathematically, this community can be characterized by a large graph in which the nodes represent people and the edges evaluate their relation strength. Because there are different kinds of relations, the edges of this graph should be heterogeneous. For some tasks, we can also model this community using several homogeneous graphs. Each graph reflects one kind of relation. Suppose an infectious disease breaks out, and the government tries to find those most likely to be infected. Obviously, the existing relationships among people cannot play an equivalent role. It seems reasonable to assume that under such a situation the relation "*works at the same place*" or "*lives together*" should play a critical role. The question becomes: "*How can we select the relation that is most relevant to the disease spreading? Is there a hidden relation (based on the explicit relations) that best reveals the spread path of the disease?*"

These questions can be modeled mathematically as **relation selection and extraction** in *multirelational social network analysis*. The problem of relation extraction can be simply stated as follows: In a heterogeneous social network, based on some labeled examples (e.g., provided by a user as queries), how can we evaluate the importance of different relations? In addition, how can we obtain a combination of the existing relations, which best matches the relation of labeled examples?

As an example, consider the network in Figure 9.18, which has three different relations, shown as (a), (b), and (c), respectively. Suppose a user requires that the four colored objects belong to the same community and specifies this with a query. Clearly, the relative importance of each of the three relations differs with respect to the user's information need. Of the three relations, we see that (a) is the most relevant to the user's need and is thus the most important, while (b) comes in second. Relation (c) can be seen as noise in regards to the user's information need. Traditional social network analysis does not distinguish these relations. The different relations are treated equally. They are simply combined together for describing the structure between objects. Unfortunately, in this example, relation (c) has a negative effect for this purpose. However, if we combine these relations according to their importance, relation (c) can be easily excluded,

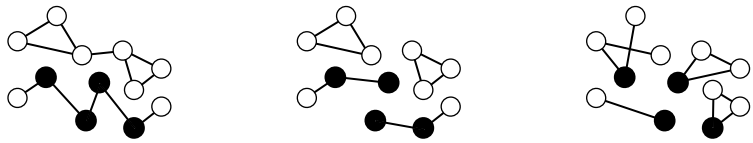

Figure 9.18 There are three relations in the network. The four colored objects are required to belong to the same community, according to a user query.

leaving relations (a) and (b) to be used to discover the community structure, which is consistent with the user's requirement.

A user might submit a more complex query in some situations. For example, a user may specify that, say, the two upper colored objects and the two lower colored objects should belong to different communities. In this situation, the importance of the three relations of Figure 9.18 changes. Relation (b) becomes the most important, while relation (a) becomes useless (and even has a negative effect with regards to the query). Thus, in a multirelational social network, community mining should be dependent on the user's query (or information need). A user's query can be very flexible. Earlier techniques focus on only a *single* relational network and are independent of the user's query and thus cannot cope with such a complex situation.

An algorithm for relation extraction and selection was proposed, which models the task as an optimization problem. The problem can be mathematically defined as follows. Given are a set of objects and a set of relations, which can be represented by a set of graphs $G_i(V, E_i)$, $i = 1, \ldots, n$, where n is the number of relations, V is the set of nodes (objects), and E_i is the set of edges with respect to the i-th relation. The weights on the edges can be naturally defined according to the relation strength of two objects. The algorithm characterizes each relation by a graph with a weight matrix. Let M_i denote the **weight matrix** associated with G_i, $i = 1, \ldots, n$. Each element in the matrix reflects the relation strength between a pair of objects in the relation. Suppose a *hidden relation* is represented by a graph $\widehat{G}(V, \widehat{E})$, and $\widehat{M}$ denotes the weight matrix associated with $\widehat{G}$. A user specifies her information need as a query in the form of a set of labeled objects $\boldsymbol{X} = [\boldsymbol{x_1}, \cdots, \boldsymbol{x_m}]$ and $\boldsymbol{y} = [y_1, \cdots, y_m]$, where y_j is the label of $\mathbf{x_j}$ (such labeled objects indicate partial information of the hidden relation $\widehat{G}$). The algorithm aims at finding a linear combination of these weight matrices that can best approximate $\widehat{G}$ (the weight matrix associated with the labeled examples.) The obtained combination is more likely to meet the user's information need, so it leads to better performance in community mining.

The algorithm was tested on bibliographic data. Naturally, multiple relations exist between authors. Authors can publish papers in thousands of different conferences, and each conference can be considered as a relation, resulting in a multirelational social network. Given some user-provided examples (like a group of authors), the algorithm can

extract a new relation using the examples and find all other groups in that relation. The extracted relation can be interpreted as the groups of authors that share certain kinds of similar interests.

9.3 Multirelational Data Mining

Relational databases are the most popular repository for *structured* data. In a relational database, multiple relations are linked together via entity-relationship links (Chapter 1). Many classification approaches (such as neural networks and support vector machines) can only be applied to data represented in single, "flat" relational form—that is, they expect data in a single table. However, many real-world applications, such as credit card fraud detection, loan applications, and biological data analysis, involve decision-making processes based on information stored in multiple relations in a relational database. Thus, multirelational data mining has become a field of strategic importance.

9.3.1 What Is Multirelational Data Mining?

Multirelational data mining (MRDM) methods search for patterns that involve multiple tables (relations) from a relational database. Consider the multirelational schema of Figure 9.19, which defines a financial database. Each table or relation represents an entity or a relationship, described by a set of attributes. Links between relations show the relationship between them. One method to apply traditional data mining methods (which assume that the data reside in a single table) is propositionalization, which converts multiple relational data into a single flat data relation, using joins and aggregations. This, however, could lead to the generation of a huge, undesirable "universal relation" (involving all of the attributes). Furthermore, it can result in the loss of information, including essential semantic information represented by the links in the database design.

Multirelational data mining aims to discover knowledge directly from relational data. There are different multirelational data mining tasks, including multirelational classification, clustering, and frequent pattern mining. Multirelational classification aims to build a classification model that utilizes information in different relations. Multirelational clustering aims to group tuples into clusters using their own attributes as well as tuples related to them in different relations. Multirelational frequent pattern mining aims at finding patterns involving interconnected items in different relations. We first use multirelational classification as an example to illustrate the purpose and procedure of multirelational data mining. We then introduce multirelational classification and multirelational clustering in detail in the following sections.

In a database for multirelational classification, there is one **target relation**, R_t, whose tuples are called **target tuples** and are associated with class labels. The other relations are *nontarget relations*. Each relation may have one *primary key* (which uniquely identifies tuples in the relation) and several *foreign keys* (where a primary key in one relation can

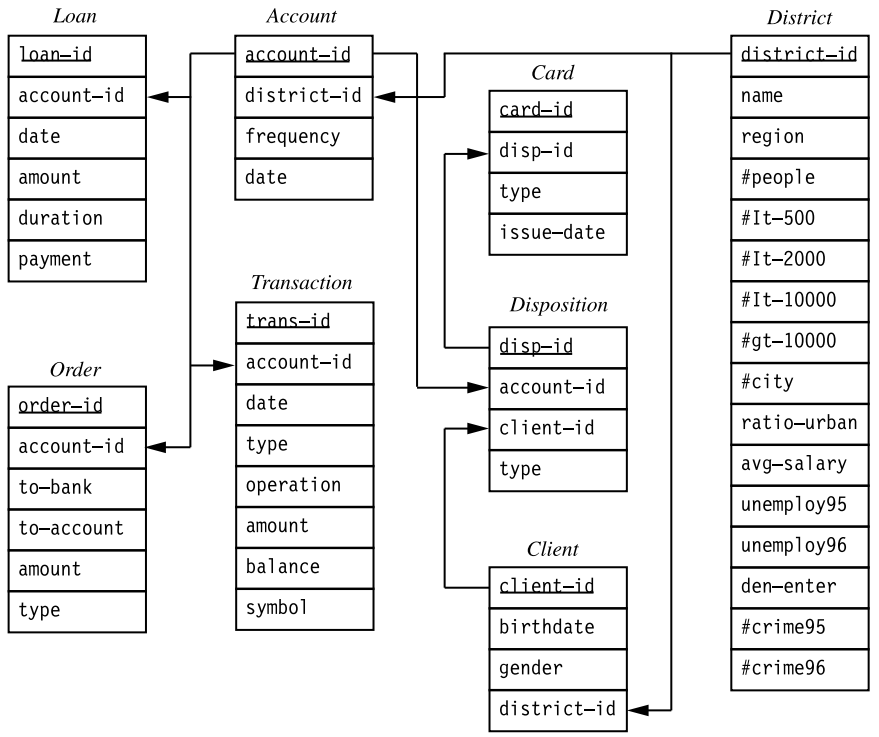

Figure 9.19 A financial database (from [PKDD CUP 99]).

be linked to the foreign key in another). If we assume a two-class problem, then we pick one class as the **positive** class and the other as the **negative** class. The most important task for building an accurate multirelational classifier is to find relevant features in different relations that help distinguish *positive* and *negative* target tuples.

Example 9.7 A database for multirelational classification. Consider the relational database of Figure 9.19. Arrows go from primary keys to corresponding foreign keys. Suppose the target relation is *Loan*. Each target tuple is either positive or negative, indicating whether the loan is paid on time. The task of multirelational classification is to build a hypothesis to distinguish positive and negative target tuples, using information in different relations. ■

For classification, in general, we search for hypotheses that help distinguish positive and negative target tuples. The most popular form of hypotheses for multirelational classification is sets of rules. Each **rule** is a list (*logical* conjunct) of *predicates*, associated with

Loan					
loan_ID	***account_ID***	*amount*	*duration*	*payment*	*class*
1	124	1000	12	120	+
2	124	4000	12	350	+
3	108	10000	24	500	−
4	45	12000	36	400	−
5	45	2000	24	90	+

Account		
account_ID	*frequency*	*date*
124	monthly	960227
108	weekly	950923
45	monthly	941209
67	weekly	950101

Figure 9.20 An example database (the last column of the *Loan* relation contains the class labels.)

a class label. A **predicate** is a constraint on an attribute in a relation. A predicate is often defined based on a certain join path, as illustrated in Example 9.8. A target tuple **satisfies** a rule if and only if it satisfies every predicate of the rule.

Example 9.8 **Predicates and rules.** *Predicate* "$p_1 = Loan(L, \_, \_, \_, payment >= 12, \_)$" means that the *duration* of loan L is no less than 12 months. It is an example of a **numerical predicate**. Predicate "$p_2 = Loan(L, A, \_, \_, \_, \_), Account(A, \_, frequency = monthly, \_)$" is defined on the *join path*, $Loan \bowtie Account$, which means that the associated account of a loan has *frequency* "*monthly*." It is a **categorical predicate**. Suppose that a rule for positive (+) target tuples is "$r = Loan(L, +) : - Loan(L, A, \_, \_, \_, \_), Account(A, \_, frequency = monthly, \_)$." We say a tuple, t, in *Loan* satisfies r if and only if **any** tuple in *Account* that is joinable with t has the value "monthly" in the *frequency* attribute of *Account*. Consider the example database of Figure 9.20. Two such tuples (namely, with *account_ID*s of 124 and 45) in *Account* satisfy the predicate $Account(A, \_, frequency = monthly, \_)$. Therefore, four tuples (with *loan_ID*s of 1, 2, 4, and 5) in *Loan* satisfy the rule. ■

Given the training data that contain a target relation and a set of nontarget relations, a rule-based multirelational classification approach will build a model consisting of a set of rules. When predicting the class label of a target tuple t, it will make the prediction based on all rules satisfied by t. For example, it may predict t's class label based on the rule with highest accuracy on the training set.

9.3.2 ILP Approach to Multirelational Classification

Inductive Logic Programming (ILP) is the most widely used category of approaches to multirelational classification. There are many ILP approaches. In general, they aim to find hypotheses of a certain format that can predict the class labels of target tuples, based on background knowledge (i.e., the information stored in all relations). The ILP problem is defined as follows: *Given background knowledge B, a set of positive examples P, and a set of negative examples N, find a hypothesis H such that: (1) $\forall t \in P : H \cup B \models t$ (completeness), and (2) $\forall t \in N : H \cup B \not\models t$ (consistency), where $\models$ stands for logical implication.*

Well-known ILP systems include FOIL, Golem, and Progol. FOIL is a top-down learner, which builds rules that cover many positive examples and few negative ones. Golem is a bottom-up learner, which performs generalizations from the most specific rules. Progol uses a combined search strategy. Recent approaches, like TILDE, Mr-SMOTI, and RPTs, use the idea of C4.5 and inductively construct decision trees from relational data.

Although many ILP approaches achieve good classification accuracy, most of them are not highly scalable with respect to the number of relations in the database. The target relation can usually join with each nontarget relation via multiple join paths. Thus, in a database with reasonably complex schema, a large number of join paths will need to be explored. In order to identify good features, many ILP approaches repeatedly join the relations along different join paths and evaluate features based on the joined relation. This is time consuming, especially when the joined relation contains many more tuples than the target relation.

We look at **FOIL** as a typical example of ILP approaches. FOIL is a sequential *covering algorithm* that builds rules one at a time. After building a rule, all positive target tuples satisfying that rule are removed, and FOIL will focus on tuples that have not been covered by any rule. When building each rule, predicates are added one by one. At each step, every possible predicate is evaluated, and the best one is appended to the current rule.

To evaluate a predicate p, FOIL temporarily appends it to the current rule. This forms the rule $r + p$. FOIL constructs a new data set, which contains all target tuples satisfying $r + p$, together with the relevant nontarget tuples on the join path specified by $r + p$. Predicate p is evaluated based on the number of positive and negative target tuples satisfying $r + p$, using the *foil gain* measure, which is defined as follows: Let $P(r)$ and $N(r)$ denote the number of positive and negative tuples satisfying a rule r, respectively. Suppose the current rule is r. The *foil gain* of p is computed as follows:

$$I(r) = -\log \frac{P(r)}{P(r) + N(r)} \tag{9.2}$$

$$foil\_gain(p) = P(r + p) \cdot [I(r) - I(r + p)] \tag{9.3}$$

Intuitively, $foil\_gain(p)$ represents the total number of bits saved in representing positive tuples by appending p to the current rule. It indicates how much the predictive power of the rule can be increased by appending p to it. The best predicate found is the one with the highest foil gain.

Example 9.9 **Search for predicates by joins.** Consider the example database of Figure 9.20. Our task is to learn rules to distinguish positive (+) and negative (−) target tuples. In order to compute the foil gain of predicates in a nontarget relation like *Account*, FOIL needs to first create a joined relation of *Loan* and *Account*, as in Figure 9.21. For each predicate p in *Account*, FOIL needs to find all positive and negative tuples satisfying $r + p$, where r is the current rule.

Loan ⋈ *Account*							
loan_ID	*account_ID*	*amount*	*duration*	*payment*	*frequency*	*date*	*class*
1	124	1000	12	120	monthly	960227	+
2	124	4000	12	350	monthly	960227	+
3	108	10000	24	500	weekly	950923	−
4	45	12000	36	400	monthly	941209	−
5	45	2000	24	90	monthly	941209	+

Figure 9.21 The joined relation of *Loan* and *Account*.

The foil gain of all predicates on a certain attribute can be computed by scanning the corresponding column in the joined relation once. It can also find the best predicate in a continuous attribute, by first sorting that column and then iterating from the smallest value to the largest one to compute the foil gain, using each value as the splitting point. ■

Many ILP approaches for multirelational classification use similar methods to evaluate predicates. For databases with complex schema, the search space is huge, and there are many possible predicates at each step. For example, in the database in Figure 9.19, *Loan* can join with *Account*, *Order*, *Transaction*, and *Disposition*, each of which can join with several other relations. To build rules, FOIL needs to repeatedly construct many joined relations by physical joins to find good predicates. This procedure becomes very time consuming for databases with reasonably complex schemas.

9.3.3 Tuple ID Propagation

Tuple ID propagation is a technique for performing virtual join, which greatly improves efficiency of multirelational classification. Instead of physically joining relations, they are virtually joined by attaching the IDs of target tuples to tuples in nontarget relations. In this way the predicates can be evaluated as if a physical join were performed. Tuple ID propagation is flexible and efficient, because IDs can easily be propagated between any two relations, requiring only small amounts of data transfer and extra storage space. By doing so, predicates in different relations can be evaluated with little redundant computation.

Suppose that the primary key of the target relation is an attribute of integers, which represents the ID of each target tuple (we can create such a primary key if there isn't one). Suppose two relations, R_1 and R_2, can be joined by attributes $R_1.A$ and $R_2.A$. In tuple ID propagation, each tuple t in R_1 is associated with a set of IDs in the target relation, represented by $\boldsymbol{IDset}(t)$. For each tuple u in R_2, we set $IDset(u) = \bigcup_{t \in R_1, t.A = u.A} IDset(t)$. That is, the tuple IDs in the *IDset* for tuple t of R_1 are propagated to each tuple, u, in R_2 that is joinable with t on attribute A.

Loan			
loan_ID	*account_ID*	···	*class*
1	124		+
2	124		+
3	108		−
4	45		−
5	45		+

Account				
account_ID	*frequency*	*date*	*ID set*	*class labels*
124	monthly	960227	1, 2	2+, 0−
108	weekly	950923	3	0+, 1−
45	monthly	941209	4, 5	1+, 1−
67	weekly	950101	–	0+, 0−

Figure 9.22 Example of tuple ID propagation (some attributes in *Loan* are not shown).

Example 9.10 **Tuple ID propagation.** Consider the example database shown in Figure 9.22, which has the same schema as in Figure 9.20. The relations are joinable on the attribute *account_ID*. Instead of performing a physical join, the IDs and class labels of target (*Loan*) tuples can be propagated to the *Account* relation. For example, the first two tuples in *Loan* are joinable with the first tuple in *Account*, thus their tuple IDs ($\{1, 2\}$) are propagated to the first tuple in *Account*. The other tuple IDs are propagated in the same way. ■

To further illustrate tuple ID propagation, let's see how it can be used to compute the foil gain of predicates without having to perform physical joins. Given relations R_1 and R_2 as above, suppose that R_1 is the target relation, and all tuples in R_1 satisfy the current rule (others have been eliminated). For convenience, let the current rule contain a predicate on $R_1.A$, which enables the join of R_1 with R_2. For each tuple u of R_2, $IDset(u)$ represents all target tuples joinable with u, using the join path specified in the current rule. If tuple IDs are propagated from R_1 to R_2, then the foil gain of every predicate in R_2 can be computed using the propagated IDs on R_2.

Example 9.11 **Computing foil gain using tuple ID propagation (IDsets).** Suppose the current rule, r, is "$Loan(L, +) : - Loan(L, A, \_, \_, \_, \_)$." From Figure 9.22, we note that three positive and two negative target tuples satisfy r. Therefore, $P(r) = 3$ and $N(r) = 2$. (This would have been determined during the process of building the current rule.) To evaluate predicate p = "$Account(A, \_, frequency = monthly, \_)$," we need to find the tuples in the *Account* relation that satisfy p. There are two such tuples, namely, $\{124, 45\}$. We find the IDs of target tuples that can be joined with these two tuples by taking the

union of their corresponding *IDsets*. This results in $\{1, 2, 4, 5\}$. Among these, there are three positive and one negative target tuples. Thus, $P(r+p) = 3$ and $N(r+p) = 1$. The foil gain of predicate p can easily be computed from Equations 9.2 and 9.3. That is, $foil\_gain(p) = 3 \cdot [-\log_2(3/5) + \log_2(3/4)] = 0.966$. Thus, with tuple propagation, we are able to compute the foil gain without having to perform any physical joins. ■

Besides propagating IDs from the target relation to relations directly joinable with it, we can also propagate IDs transitively from one nontarget relation to another. Suppose two nontarget relations, R_2 and R_3, can be joined by attributes $R_2.A$ and $R_3.A$. For each tuple v in R_2, $IDset(v)$ represents the target tuples joinable with v (using the join path specified by the current rule). By propagating IDs from R_2 to R_3 through the join $R_2.A = R_3.A$, for each tuple u in R_3, $IDset(u)$ represents target tuples that can be joined with u (using the join path in the current rule, plus the join $R_2.A = R_3.A$). Thus, by tuple ID propagation between nontarget relations, we can also compute the foil gain based on the propagated IDs.

Tuple ID propagation, although valuable, should be enforced with certain constraints. There are two cases where such propagation could be counterproductive: (1) propagation via large fan-outs, and (2) propagation via long, weak links. The first case occurs when, after propagating the IDs to a relation R, it is found that every tuple in R is joined with many target tuples and every target tuple is joined with many tuples in R. The semantic link between R and the target relation is then typically very weak because the link is unselective. For example, propagation among people via birth-country links may not be productive. The second case occurs when the propagation goes through long links (e.g., linking a student with his car dealer's pet may not be productive, either). From the sake of efficiency and accuracy, propagation via such links is discouraged.

9.3.4 Multirelational Classification Using Tuple ID Propagation

In this section we introduce **CrossMine**, an approach that uses tuple ID propagation for multirelational classification. To better integrate the information of ID propagation, CrossMine uses *complex predicates* as elements of rules. A complex predicate, p, contains two parts:

1. *prop-path*: This indicates how to propagate IDs. For example, the path "*Loan.account_ID* $\rightarrow$ *Account.account_ID*" indicates propagating IDs from *Loan* to *Account* using *account_ID*. If no ID propagation is involved, *prop-path* is empty.
2. *constraint*: This is a predicate indicating the constraint on the relation to which the IDs are propagated. It can be either categorical or numerical.

A complex predicate is usually equivalent to two conventional predicates. For example, the rule "$Loan(L, +) : -Loan(L, A, \_, \_, \_, \_), Account(A, \_, frequent = monthly, \_)$" can be represented by "$Loan(+) : -[Loan.account\_ID \rightarrow Account.account\_ID, Account.frequency = monthly]$."

CrossMine builds a classifier containing a set of rules, each containing a list of complex predicates and a class label. The algorithm of CrossMine is shown in Figure 9.23. CrossMine is also a sequential covering algorithm like FOIL. It builds rules one at a time. After a rule r is built, all positive target tuples satisfying r are removed from the data

Algorithm: CrossMine. Rule-based classification across multiple relations.

Input:

- D, a relational database;
- R_t a target relation.

Output:

- A set of rules for predicting class labels of target tuples.

Method:

```
(1)  rule set R ← ∅;
(2)  while (true)
(3)    rule r ← empty-rule;
(4)    set R_t to active;
(5)    repeat
(6)      Complex predicate p ← the predicate with highest foil gain;
(7)      if foil_gain(p) < MIN_FOIL_GAIN then
(8)        break;
(9)      else
(10)       r ← r + p; // append predicate, increasing rule length by 1
(11)       remove all target tuples not satisfying r;
(12)       update IDs on every active relation;
(13)       if p.constraint is on an inactive relation then
(14)         set that relation active;
(15)     endif
(16)   until (r.length = MAX_RULE_LENGTH)
(17)   if r = empty-rule then break;
(18)   R ← R ∪ {r};
(19)   remove all positive target tuples satisfying r;
(20)   set all relations inactive;
(21) endwhile
(22) return R;
```

Figure 9.23 Algorithm CrossMine.

set. To build a rule, CrossMine repeatedly searches for the best complex predicate and appends it to the current rule, until the stop criterion is met. A relation is active if it appears in the current rule. Before searching for the next best predicate, each active relation is required to have the *IDset* of propagated IDs for each of its tuples. When searching for a predicate, CrossMine evaluates all of the possible predicates on any active relation or any relation that is joinable with an active relation. When there are more than two classes of target tuples, CrossMine builds a set of rules for each class.

"*How does CrossMine find the best predicate to append to the current rule?*" At each step, CrossMine uses tuple ID propagation to search for the best predicate in all of the active relations, or relations that are joinable with any active relation. In our example, at first only the *Loan* relation is active. If the first best predicate is found in, say, the *Account* relation, *Account* becomes active as well. CrossMine tries to propagate the tuple IDs from *Loan* or *Account* to other relations to find the next best predicate. In this way, the search range is gradually expanded along promising directions reflecting strong semantic links between entities. This avoids aimless search in the huge hypothesis space.

Suppose that CrossMine is searching for the best predicate on a categorical attribute, A_c, in a relation, R. CrossMine evaluates all possible predicates and selects the best one. For each value a_i of A_c, a predicate $p_i = [R.A_c = a_i]$ is built. CrossMine scans the values of each tuple on A_c to find the numbers of positive and negative target tuples satisfying each predicate, p_i. The foil gain of each p_i can then be computed to find the best predicate. For numerical predicates, it uses the method described in Section 9.3.2.

The above algorithm may fail to find good predicates in databases containing relations that are only used to join with other relations. For example, in the database of Figure 9.24, there is no useful attribute in the *Has_Loan* relation. Therefore, the rules built will not involve any predicates on the *Client* and *District* relations. CrossMine adopts a *look-one-ahead* method to solve this problem. After IDs have been propagated to a relation $\bar{R}$ (such as *Has_Loan*), if $\bar{R}$ contains a foreign key referencing the primary key of a relation $\bar{R}'$ (such as *client_ID* of *Client*), then IDs are propagated from $\bar{R}$ to $\bar{R}'$, and used

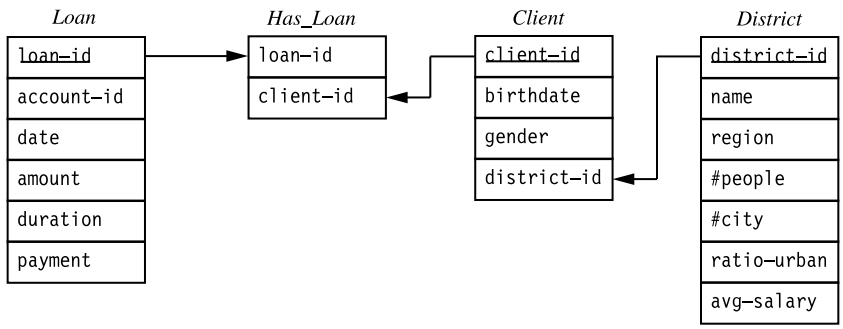

Figure 9.24 Another example database.

to search for good predicates in $\bar{R}'$. By this method, in the example of Figure 9.24, we can find rules like "$Loan(+) : -[Loan.loan\_ID \rightarrow Has\_Loan.loan\_ID, Has\_Loan.client\_ID \rightarrow Client.client\_ID, Client.birthdate < 01/01/60]$."

After generating a classifier, CrossMine needs to predict the class labels of unlabeled target tuples. Suppose there is a rule, $r = R_t(+) : -p_1, p_2, \ldots, p_k$, where each p_i is a complex predicate. CrossMine propagates the IDs of target tuples along the *prop-path* of each predicate, and prunes all IDs of target tuples not satisfying the constraint of p_i. In this way it can easily find all target tuples satisfying each rule. For each target tuple t, the most accurate rule that is satisfied by t is used to predict the class label of t.

"*How does CrossMine fare in terms of scalability and accuracy?*" Experiments have shown that CrossMine is highly scalable compared with traditional ILP approaches and also achieves high accuracy. These features make it attractive for multirelational classification in real-world databases.

9.3.5 Multirelational Clustering with User Guidance

Multirelational clustering is the process of partitioning data objects into a set of clusters based on their similarity, utilizing information in multiple relations. In this section we will introduce **CrossClus** (Cross-relational Clustering with user guidance), an algorithm for multirelational clustering that explores how to utilize user guidance in clustering and tuple ID propagation to avoid physical joins.

One major challenge in multirelational clustering is that there are too many attributes in different relations, and usually only a small portion of them are relevant to a specific clustering task. Consider the computer science department database of Figure 9.25. In order to cluster students, attributes cover many different aspects of information, such as courses taken by students, publications of students, advisors and research groups of students, and so on. A user is usually interested in clustering students using a certain aspect of information (e.g., clustering students by their research areas). Users often have a good grasp of their application's requirements and data semantics. Therefore, a user's guidance, even in the form of a simple query, can be used to improve the efficiency and quality of high-dimensional multirelational clustering. CrossClus accepts user queries that contain a *target relation* and one or more *pertinent attributes*, which together specify the clustering goal of the user.

Example 9.12 **User guidance in the form of a simple query.** Consider the database of Figure 9.25. Suppose the user is interested in clustering students based on their research areas. Here, the target relation is *Student* and the pertinent attribute is *area* from the *Group* relation. A user query for this task can be specified as "cluster *Student* with *Group.area*." ■

In order to utilize attributes in different relations for clustering, CrossClus defines *multirelational attributes*. A **multirelational attribute** $\tilde{A}$ is defined by a join path $R_t \bowtie R_1 \bowtie \cdots \bowtie R_k$, an attribute $R_k.A$ of R_k, and possibly an aggregation operator (e.g., average, count, max). $\tilde{A}$ is formally represented by $[\tilde{A}.joinpath, \tilde{A}.attr, \tilde{A}.aggr]$, in which $\tilde{A}.aggr$ is optional. A multirelational attribute $\tilde{A}$ is either a *categorical feature*

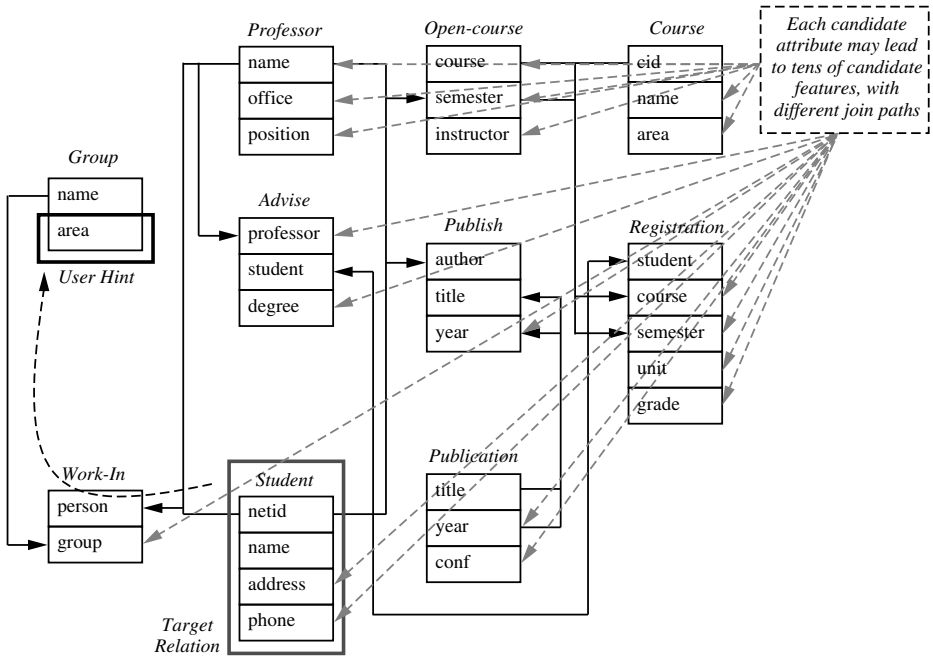

Figure 9.25 Schema of a computer science department database.

or a *numerical one*, depending on whether $R_k.A$ is categorical or numerical. If $\tilde{A}$ is a categorical feature, then for a target tuple t, $t.\tilde{A}$ represents the distribution of values among tuples in R_k that are joinable with t. For example, suppose $\tilde{A} = [Student \bowtie Register \bowtie OpenCourse \bowtie Course, area]$ (areas of courses taken by each student). If a student t_1 takes four courses in database and four courses in AI, then $t_1.\tilde{A} =$ (database:0.5, AI:0.5). If $\tilde{A}$ is numerical, then it has a certain aggregation operator (average, count, max, ...), and $t.\tilde{A}$ is the aggregated value of tuples in R_k that are joinable with t.

In the multirelational clustering process, CrossClus needs to search pertinent attributes across multiple relations. CrossClus must address two major challenges in the searching process. First, the target relation, R_t, can usually join with each nontarget relation, R, via many different join paths, and each attribute in R can be used as a multirelational attribute. It is impossible to perform any kind of exhaustive search in this huge search space. Second, among the huge number of attributes, some are pertinent to the user query (e.g., a student's advisor is related to her research area), whereas many others are irrelevant (e.g., a student's classmates' personal information). How can we identify pertinent attributes while avoiding aimless search in irrelevant regions in the attribute space?

To overcome these challenges, CrossClus must confine the search process. It considers the relational schema as a graph, with relations being nodes and joins being edges. It adopts a heuristic approach, which starts search from the user-specified attribute, and then repeatedly searches for useful attributes in the neighborhood of existing attributes. In this way it gradually expands the search scope to related relations, but will not go deep into random directions.

"*How does CrossClus decide if a neighboring attribute is pertinent?*" CrossClus looks at how attributes cluster target tuples. The pertinent attributes are selected based on their relationships to the user-specified attributes. In essence, if two attributes cluster tuples very differently, their similarity is low and they are unlikely to be related. If they cluster tuples in a similar way, they should be considered related. However, if they cluster tuples in almost the same way, their similarity is very high, which indicates that they contain redundant information. From the set of pertinent features found, CrossClus selects a set of nonredundant features so that the similarity between any two features is no greater than a specified maximum.

CrossClus uses the **similarity vector** of each attribute for evaluating the similarity between attributes, which is defined as follows. Suppose there are N target tuples, $t_1, \ldots, t_N$. Let $\mathbf{V}^{\tilde{A}}$ be the similarity vector of attribute $\tilde{A}$. It is an N^2-dimensional vector that indicates the similarity between each pair of target tuples, t_i and t_j, based on $\tilde{A}$. To compare two attributes by the way they cluster tuples, we can look at how alike their similarity vectors are, by computing the inner product of the two similarity vectors. However, this is expensive to compute. Many applications cannot even afford to store N^2-dimensional vectors. Instead, CrossClus converts the hard problem of computing the similarity between similarity vectors to an easier problem of computing *similarities between attribute values*, which can be solved in linear time.

Example 9.13 **Multirelational search for pertinent attributes.** Let's look at how CrossClus proceeds in answering the query of Example 9.12, where the user has specified her desire to cluster students by their research areas. To create the initial multirelational attribute for this query, CrossClus searches for the shortest join path from the target relation, *Student*, to the relation *Group*, and creates a multirelational attribute $\tilde{A}$ using this path. We simulate the procedure of attribute searching, as shown in Figure 9.26. An initial pertinent multirelational attribute [*Student* $\bowtie$ *WorkIn* $\bowtie$ *Group* , *area*] is created for this query (step 1 in the figure). At first CrossClus considers attributes in the following relations that are joinable with either the target relation or the relation containing the initial pertinent attribute: *Advise*, *Publish*, *Registration*, *WorkIn*, and *Group*. Suppose the best attribute is [*Student* $\bowtie$ *Advise* , *professor*], which corresponds to the student's advisor (step 2). This brings the *Professor* relation into consideration in further search. CrossClus will search for additional pertinent features until most tuples are sufficiently covered. CrossClus uses tuple ID propagation (Section 9.3.3) to virtually join different relations, thereby avoiding expensive physical joins during its search. ■

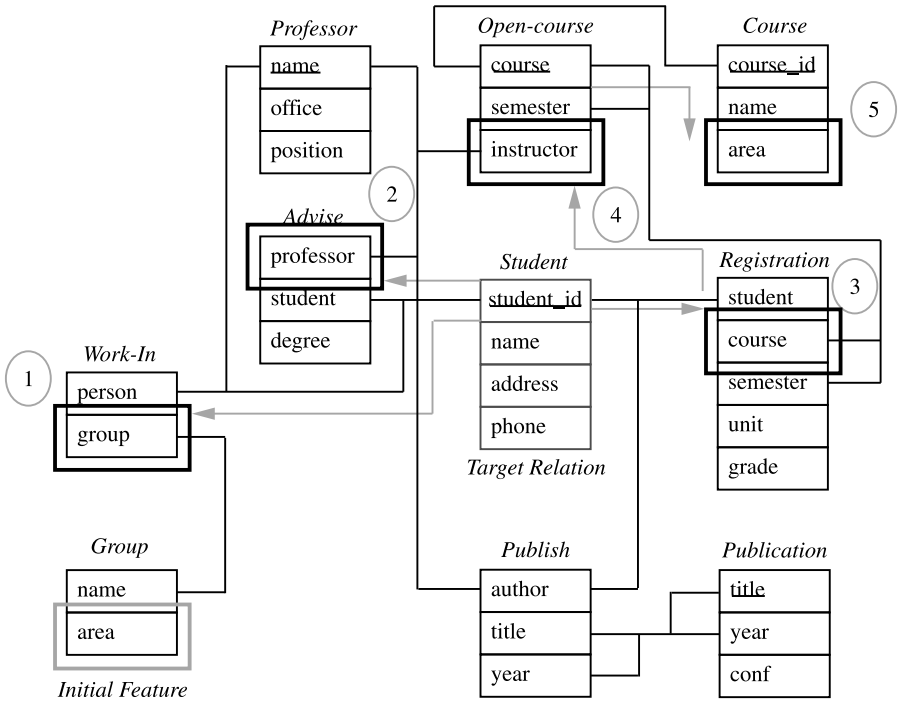

Figure 9.26 Search for pertinent attributes in CrossClus.

Now that we have an intuitive idea of how CrossClus employs user guidance to search for attributes that are highly pertinent to the user's query, the next question is, how does it perform the actual clustering? With the potentially large number of target tuples, an efficient and scalable clustering algorithm is needed. Because the multirelational attributes do not form a Euclidean space, the k-medoids method (Section 7.4.1) was chosen, which requires only a distance measure between tuples. In particular, CLARANS (Section 7.4.2), an efficient k-medoids algorithm for large databases, was used. The main idea of CLARANS is to consider the whole space of all possible clusterings as a graph and to use randomized search to find good clusterings in this graph. It starts by randomly selecting k tuples as the initial medoids (or cluster representatives), from which it constructs k clusters. In each step, an existing medoid is replaced by a new randomly selected medoid. If the replacement leads to better clustering, the new medoid is kept. This procedure is repeated until the clusters remain stable.

CrossClus provides the clustering results to the user, together with information about each attribute. From the attributes of multiple relations, their join paths, and aggregation operators, the user learns the meaning of each cluster, and thus gains a better understanding of the clustering results.

9.4 Summary

- **Graphs** represent a more general class of structures than sets, sequences, lattices, and trees. **Graph mining** is used to mine frequent graph patterns, and perform characterization, discrimination, classification, and cluster analysis over large graph data sets. Graph mining has a broad spectrum of applications in chemical informatics, bioinformatics, computer vision, video indexing, text retrieval, and Web analysis.
- Efficient methods have been developed for **mining frequent subgraph patterns**. They can be categorized into Apriori-based and pattern growth–based approaches. The *Apriori-based approach* has to use the breadth-first search (BFS) strategy because of its level-wise candidate generation. The *pattern-growth approach* is more flexible with respect to the search method. A typical pattern-growth method is gSpan, which explores additional optimization techniques in pattern growth and achieves high performance. The further extension of gSpan for mining closed frequent graph patterns leads to the CloseGraph algorithm, which mines more compressed but complete sets of graph patterns, given the minimum support threshold.
- There are many interesting **variant graph patterns**, including approximate frequent graphs, coherent graphs, and dense graphs. A general framework that considers constraints is needed for mining such patterns. Moreover, various user-specific constraints can be pushed deep into the graph pattern mining process to improve mining efficiency.
- Application development of graph mining has led to the generation of compact and effective **graph index structures** using frequent and discriminative graph patterns. **Structure similarity search** can be achieved by exploration of multiple graph features. **Classification and cluster analysis of graph data sets** can be explored by their integration with the graph pattern mining process.
- A **social network** is a *heterogeneous* and *multirelational* data set represented by a graph, which is typically very large, with *nodes* corresponding to *objects*, and *edges* (or *links*) representing relationships between objects.
- **Small world networks** reflect the concept of small worlds, which originally focused on networks among individuals. They have been characterized as having a high degree of local clustering for a small fraction of the nodes (i.e., these nodes are interconnected with one another), while being no more than a few degrees of separation from the remaining nodes.
- Social networks exhibit certain characteristics. They tend to follow the **densification power law**, which states that networks become increasingly *dense* over time. **Shrinking diameter** is another characteristic, where the effective diameter often *decreases* as the network grows. Node *out-degrees* and *in-degrees* typically follow a **heavy-tailed distribution**. A **Forest Fire model** for graph generation was proposed, which incorporates these characteristics.

- **Link mining** is a confluence of research in social networks, link analysis, hypertext and Web mining, graph mining, relational learning, and inductive logic programming. Link mining tasks include *link-based object classification*, *object type prediction*, *link type prediction, link existence prediction*, *link cardinality estimation*, *object reconciliation* (which predicts whether two objects are, in fact, the same), and *group detection* (which clusters objects). Other tasks include *subgraph identification* (which finds characteristic subgraphs within networks) and *metadata mining* (which uncovers schema-type information regarding unstructured data).
- In **link prediction**, measures for analyzing the proximity of network nodes can be used to predict and rank new links. Examples include the *shortest path* (which ranks node pairs by their shortest path in the network) and *common neighbors* (where the greater the number of neighbors that two nodes share, the more likely they are to form a link). Other measures may be based on the *ensemble* of all paths between two nodes.
- **Viral marketing** aims to optimize the positive word-of-mouth effect among customers. By considering the interactions between customers, it can choose to spend more money marketing to an individual if that person has many social connections.
- Newsgroup discussions form a kind of network based on the "responded-to" relationships. Because people generally respond more frequently to a message when they *disagree* than when they *agree*, graph partitioning algorithms can be used to **mine newsgroups** based on such a network to effectively classify authors in the newsgroup into *opposite camps*.
- Most community mining methods assume that there is only one kind of relation in the network, and moreover, the mining results are independent of the users' information needs. In reality, there may be multiple relations between objects, which collectively form a **multirelational social network** (or *heterogeneous social network*). *Relation selection and extraction* in such networks evaluates the importance of the different relations with respect to user information provided as queries. In addition, it searches for a combination of the existing relations that may reveal a *hidden community* within the multirelational network.
- **Multirelational data mining** (**MRDM**) methods search for patterns that involve *multiple tables* (relations) from a relational database.
- **Inductive Logic Programming** (**ILP**) is the most widely used category of approaches to multirelational classification. It finds hypotheses of a certain format that can predict the class labels of target tuples, based on background knowledge. Although many ILP approaches achieve good classification accuracy, most are not highly scalable due to the computational expense of repeated joins.
- **Tuple ID propagation** is a method for virtually joining different relations by attaching the IDs of target tuples to tuples in nontarget relations. It is much less costly than physical joins, in both time and space.

- 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, "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 *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] proposed 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, were described in Liben-Nowell and Kleinberg [LNK03]. The Katz measure was presented in Katz [Kat53]. A probabilistic model for learning link structure was 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] provided 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 construct 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] proposed an approach to integrate ILP and statistical modeling for document classification and retrieval. The CrossMine approach was 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].