Chapter 6

Advanced Pattern Mining

Frequent pattern mining has reached far beyond the basics due to substantial research, numerous extensions of the problem scope, and broad application studies. In this chapter, we will learn methods for advanced pattern mining. We first introduce methods for mining various kinds of patterns, including mining multilevel patterns, multidimensional patterns, patterns in continuous data, rare patterns, negative patterns, and frequent patterns in high-dimensional data. We also examine methods for mining compressed and approximate patterns. Then we examine the methodologies of utilizing constraints to reduce the cost of frequent pattern mining. Since sequential patterns and structural patterns are popularly encountered but they need rather different mining methods, we introduce concepts and methods for mining sequential patterns in sequence datasets and mining subgraph patterns in graph datasets. To get the flavor on how to extend pattern mining methods to facilitate diverse applications, we examine one example on mining copy-and-paste bugs in large software programs. Notice that *pattern mining* is a more general term than *frequent pattern mining* since the former covers rare and negative patterns as well. However, when there is no ambiguity, the two terms are used interchangeably.

6.1 Mining Various Kinds of Patterns

In the last chapter we have studied methods for mining patterns and associations at a single concept level and single dimensional space (e.g., products purchased). However, in many applications, people may like to uncover more complex patterns from massive data. For example, one may like to find *multilevel associations* which involve concepts at different abstraction levels, *multidimensional associations* which involve more than one dimension or predicate (e.g., rules that relate what a customer *buys* to his or her *age*), *quantitative association rules* that involve numeric attributes (e.g., *age, salary*), *rare patterns* which may suggest interesting although rare item combinations, and *negative patterns* that show negative correlations between items.

In this section we examine methods for mining patterns and associations at multiple abstraction levels (Section 6.1.1) and at multi-dimensional spaces (Section 6.1.2), handling data with quantitative attributes (Section 6.1.3), mining patterns in high-dimensional space (Section 6.1.4), and mining rare patterns and negative patterns (Section 6.1.5).

6.1.1 Mining Multilevel Associations

For many applications, strong associations discovered at high abstraction levels, though often having high support, could be commonsense knowledge (e.g., buying bread and milk frequently together). We may want to drill down to find novel patterns at more detailed levels (e.g., buying what kind of bread and what kind of milk frequently together). On the other hand, there could be too many scattered patterns at low or primitive abstraction levels, some of which are just trivial specializations of patterns at higher levels. Therefore, it is interesting to examine how to develop effective methods for mining meaningful patterns at multiple abstraction levels, with sufficient flexibility for easy traversal among different abstraction spaces.

Example 6.1.1 **Mining multilevel association rules.** Suppose we are given the task-relevant set of transactional data in Table 6.1 for sales in a store, showing the items purchased for each transaction. The concept hierarchy for the items is shown in Figure 6.1. A concept hierarchy defines a sequence of mappings from a set of low-level concepts to a higher-level, more general concept set. Data can be generalized by replacing low-level concepts within the data by their corresponding higher-level concepts, or *ancestors*, from a concept hierarchy.

The concept hierarchy in Figure 6.1 has five levels, respectively referred to as levels 0 through 4, starting with level 0 at the root node for all (the most general abstraction level). Here, level 1 includes *computer, software, printer and camera,* and *computer accessory*; level 2 includes *laptop computer, desktop computer*, *office software, antivirus software*, etc.; and level 3 includes *Dell desktop computer, . . . , Microsoft office software*, etc. Level 4 is the most specific abstraction level of this hierarchy. It consists of concrete products.

Concept hierarchies for nominal attributes may be specified by users familiar with the data such as store managers. Alternatively, they can be generated from data, based on the analysis of product specifications, attribute values, or data distributions. Concept hierarchies for numeric attributes can be generated using discretization techniques, such as those introduced in Chapter 3. For our example, the concept hierarchy of Figure 6.1 is provided.

The items in Table 6.1 are at the lowest level of Figure 6.1's concept hierarchy. It is difficult to find interesting purchase patterns in such primitive-level data. For instance, if "*Dell Studio XPS 16 Notebook*" or "*Logitech VX Nano Cordless Laser Mouse*" occurs in a very small fraction of the transactions, then it can be difficult to find strong associations involving these specific items. Few people may buy these items together, making it unlikely that the itemset will satisfy minimum support. However, we would expect that it is easier to find

Table 6.1: Task-Relevant Data, D

TID	*Items Purchased*
T100	Apple 15″ MacBook Pro, HP Photosmart 7520 printer
T200	Microsoft Office Professional 2020, Microsoft Surface Mobile Mouse
T300	Logitech MX Master 2S Wireless Mouse, Fellowes GEL Wrist Rest
T400	Dell Studio XPS 16 Notebook, Canon PowerShot SX70 HS Digital Camera
T500	Apple iPad Air (10.5-inch, Wi-Fi, 256GB), Norton Security Premium
...	...

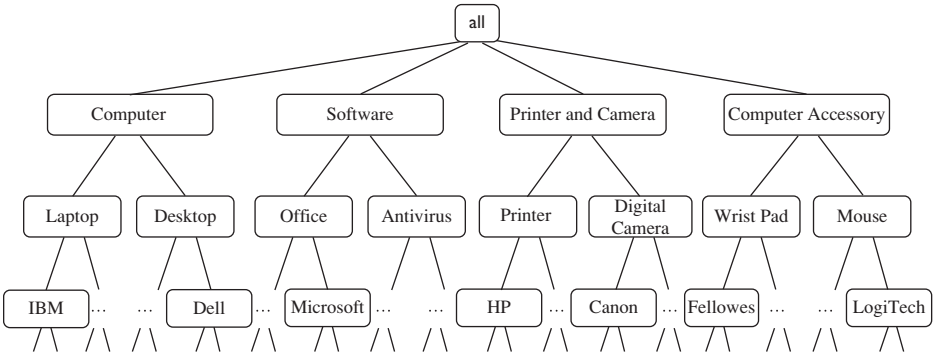

Figure 6.1: Concept hierarchy for *AllElectronics* computer items.

strong associations between generalized abstractions of these items, such as between "*Dell Notebook*" and "*Cordless Mouse.*"

Association rules generated from mining data at multiple abstraction levels are called **multiple-level** or **multilevel association rules**. Multilevel association rules can be mined efficiently using concept hierarchies under a support-confidence framework. In general, a top-down strategy can be employed, where counts are accumulated for the calculation of frequent itemsets at each concept level, starting at concept level 1 and working downward in the hierarchy toward the more specific concept levels, until no more frequent itemsets can be found. For each level, any algorithm for discovering frequent itemsets may be used, such as Apriori or its variations.

A number of variations to this approach are described next, where each variation involves "playing" with the support threshold in a slightly different way. The variations are illustrated in Figures 6.2 and 6.3, where nodes indicate an item or itemset that has been examined, and nodes with thick borders indicate that an examined item or itemset is frequent.

- **Using uniform minimum support for all levels** (referred to as **uniform support**): The same minimum support threshold is used when mining at each abstraction level. For example, in Figure 6.2, a minimum

support threshold of 5% is used throughout (e.g., for mining from "*computer*" downward to "*laptop computer*"). Both "*computer*" and "*laptop computer*" are found to be frequent, whereas "*desktop computer*" is not. When a uniform minimum support threshold is used, the search procedure is simplified. The method is also simple in that users are required to specify only one minimum support threshold. An Apriori-like optimization technique can be adopted, based on the knowledge that an ancestor is a superset of its descendants: The search avoids examining itemsets containing any item or itemset of which the ancestors do not have minimum support.

The uniform support approach, however, has some drawbacks. It is unlikely that items at lower abstraction levels will occur as frequently as those at higher abstraction levels. If the minimum support threshold is set too high, it could miss some meaningful associations occurring at low abstraction levels. If the threshold is set too low, it may generate many uninteresting associations occurring at high abstraction levels. This provides the motivation for the next approach.

- **Using reduced minimum support at lower levels** (referred to as **reduced support**): Each abstraction level has its own minimum support threshold. The deeper the abstraction level, the smaller the corresponding threshold. For example, in Figure 6.3, the minimum support thresholds for levels 1 and 2 are 5% and 3%, respectively. In this way, "*computer,*" "*laptop computer,*" and "*desktop computer*" are all considered frequent.
  For mining multilevel patterns with reduced support, the minimum support threshold at the lowest abstraction level should be used during the mining process to allow mining to penetrate down to the lowest abstraction level. However, for the final pattern/rule extraction, thresholds associated with the corresponding items should be enforced to print out only interesting associations.

- **Using item or group-based minimum support (referred to as group-based support)**: Because users or experts often have insight as to which groups are more important than others, it is sometimes desir-

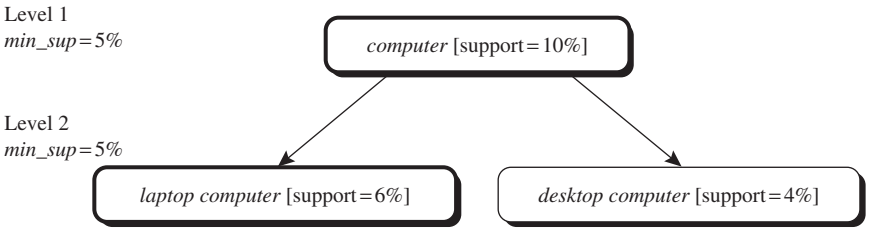

Figure 6.2: Multilevel mining with uniform support.

able to set up user-specific, item-based, or group-based minimal support thresholds when mining multilevel rules. For example, a user could set up the minimum support thresholds based on product price or on items of interest, such as by setting particularly low support thresholds for "*camera with price over $ 1000*", to pay particular attention to the association patterns containing items in these categories.
For mining patterns with mixed items from groups with different support thresholds, usually the lowest support threshold among all the participating groups is taken as the support threshold in mining. This will avoid filtering out valuable patterns containing items from the group with the lowest support threshold. In the meantime, the minimal support threshold for each individual group should be kept to avoid generating uninteresting itemsets from each group. Other interestingness measures can be used after the itemset mining to extract truly interesting rules.

A serious side effect of mining multilevel association rules is its generation of many redundant rules across multiple abstraction levels due to the "ancestor" relationships among items. For example, consider the following rules where "*laptop computer*" is an ancestor of "*Dell laptop computer*" based on the concept hierarchy of Figure 6.1, and where X is a variable representing customers who purchased items.

$$buys(X, \text{“}laptop\ computer\text{”}) \Rightarrow buys(X, \text{“}HP\ printer\text{”}) \\ [support = 8\%, confidence = 70\%] \quad (6.1)$$

$$buys(X, \text{“}Dell\ laptop\ computer\text{”}) \Rightarrow buys(X, \text{“}HP\ printer\text{”}) \\ [support = 2\%, confidence = 72\%] \quad (6.2)$$

"*If Rules (6.1) and (6.2) are both mined, does Rule (6.2) provide any novel information?*" We say a rule $R1$ is an **ancestor** of a rule $R2$, if $R1$ can be obtained by replacing the items in $R2$ by their ancestors in a concept hierarchy. For example, Rule (6.1) is an ancestor of Rule (6.2) because "*laptop computer*" is an ancestor of "*Dell laptop computer.*" Based on this definition, a rule can be

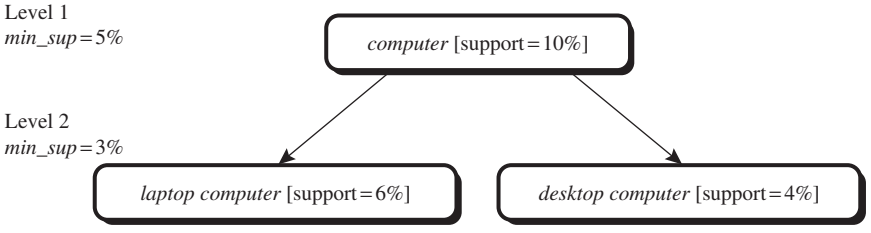

Figure 6.3: Multilevel mining with reduced support.

considered redundant if its support and confidence are close to their "expected" values, based on an ancestor of the rule.

Example 6.1.2 **Checking redundancy among multilevel association rules.** Suppose that about one-quarter of all "*laptop computer*" sales are for "*Dell laptop computers*." Since Rule (6.1) has a 70% confidence and 8% support, we may expect Rule (6.2) to have a confidence of around 70% (since all data samples of "*Dell laptop computer*" are also samples of "*laptop computer*") and a support of around 2% (i.e., $8\% \times \frac{1}{4}$). If this is indeed the case, then Rule (6.2) is not interesting because it does not offer any additional information and is less general than Rule (6.1).

6.1.2 Mining Multidimensional Associations

So far, we have studied association rules that imply a single predicate, that is, the predicate *buys*. For instance, in mining our *AllElectronics* database, we may discover the Boolean association rule

$$buys(X, \text{“}Apple\ iPad\ air\text{”}) \Rightarrow buys(X, \text{“}HP\ printer\text{”}). \quad (6.3)$$

Following the terminology used in multidimensional databases, we refer to each distinct predicate in a rule as a dimension. Hence, we can refer to Rule (6.3) as a **single-dimensional** or **intradimensional association rule** because it contains a single distinct predicate (e.g., *buys*) with multiple occurrences (i.e., the predicate occurs more than once within the rule). Such rules are commonly mined from transactional data.

Instead of considering transactional data only, sales and related information are often linked with relational data or integrated into a data warehouse. Such data stores are multidimensional in nature. For instance, in addition to keeping track of the items purchased in sales transactions, a relational database may record other attributes associated with the items and/or transactions such as the item description or the branch location of the sale. Additional relational information regarding the customers who purchased the items (e.g., customer age, occupation, credit rating, income, and address) may also be stored. Considering each database attribute or warehouse dimension as a predicate, we can therefore mine association rules containing *multiple* predicates such as

$$age(X, \text{“}18 \ldots 25\text{”}) \wedge occupation(X, \text{“}student\text{”}) \Rightarrow buys(X, \text{“}laptop\text{”}). \quad (6.4)$$

Association rules that involve two or more dimensions or predicates can be referred to as **multidimensional association rules**. Rule (6.4) contains three predicates (*age, occupation*, and *buys*), each of which occurs *only once* in the rule. Hence, we say that it has **no repeated predicates**. Multidimensional association rules with no repeated predicates are called **interdimensional association rules**. We can also mine multidimensional association rules with repeated predicates, which contain multiple occurrences of some predicates. These

rules are called **hybrid-dimensional association rules**. An example of such a rule is the following, where the predicate *buys* is repeated:

$$age(X, \text{“18}\ldots\text{25”}) \wedge buys(X, \text{“}laptop\text{”}) \Rightarrow buys(X, \text{“}HP\ printer\text{”}). \quad (6.5)$$

Database attributes can be nominal or quantitative. The values of **nominal** (or categorical) attributes are “names of things.” Nominal attributes have a finite number of possible values, with no ordering among the values (e.g., *occupation, brand*, *color*). **Quantitative** attributes are numeric and have an implicit ordering among values (e.g., *age, income*, *price*). Techniques for mining multidimensional association rules can be categorized into two basic approaches regarding the treatment of quantitative attributes.

In the first approach, *quantitative attributes are discretized using predefined concept hierarchies*. This discretization occurs before mining. For instance, a concept hierarchy for *income* may be used to replace the original numeric values of this attribute by interval labels such as “0..20K,” “21K..30K,” “31K..40K,” and so on. Here, discretization is *static* and predetermined. Chapter 3 on data preprocessing gave several techniques for discretizing numeric attributes. The discretized numeric attributes, with their interval labels, can then be treated as nominal attributes (where each interval is considered a category). We refer to this as **mining multidimensional association rules using static discretization of quantitative attributes**.

In the second approach, *quantitative attributes are discretized or clustered into “bins” based on the data distribution*. These bins may be further combined during the mining process. The discretization process is *dynamic* and established so as to satisfy some mining criteria such as maximizing the confidence of the rules mined. Because this strategy treats the numeric attribute values as quantities rather than as predefined ranges or categories, association rules mined from this approach are also referred to as **(dynamic) quantitative association rules**.

Let’s study each of these approaches for mining multidimensional association rules. For simplicity, we confine our discussion to interdimensional association rules. Note that rather than searching for frequent itemsets (as is done for single-dimensional association rule mining), in multidimensional association rule mining we search for frequent *predicate sets*. A ***k*-predicate set** is a set containing k conjunctive predicates. For instance, the set of predicates {*age, occupation, buys*} from Rule (6.4) is a 3-predicate set.

6.1.3 Mining Quantitative Association Rules

As discussed earlier, relational and data warehouse data often involve quantitative attributes or measures. We can discretize quantitative attributes into multiple intervals and then treat them as nominal data in association mining. However, such simple discretization may lead to the generation of an enormous number of rules, many of which may not be useful. Here we introduce three methods that can help overcome this difficulty to discover novel association re-

lationships: (1) a data cube method, (2) a clustering-based method, and (3) a statistical analysis method to uncover exceptional behaviors.

Data Cube-Based Mining of Quantitative Associations

In many cases quantitative attributes can be discretized before mining using predefined concept hierarchies or data discretization techniques, where numeric values are replaced by interval labels. Nominal attributes may also be generalized to higher conceptual levels if desired. If the resulting task-relevant data are stored in a relational table, then any of the frequent itemset mining algorithms we have discussed can easily be modified so as to find all frequent predicate sets. In particular, instead of searching on only one attribute like *buys*, we need to search through all of the relevant attributes, treating each attribute-value pair as an itemset.

Alternatively, the transformed multidimensional data may be used to construct a *data cube*. Data cubes are well suited for the mining of multidimensional association rules: They store aggregates (e.g., counts) in multidimensional space, which is essential for computing the support and confidence of multidimensional association rules. An overview of data cube technology was presented in Chapter 4. Detailed algorithms for data cube computation were given in Chapter 5. Figure 6.4 shows the lattice of cuboids defining a data cube for the dimensions *age, income*, and *buys*. The cells of an n-dimensional cuboid can be used to store the support counts of the corresponding n-predicate sets. The base cuboid aggregates the task-relevant data by *age, income*, and *buys*; the 2-D cuboid, (*age, income*), aggregates by *age* and *income*, and so on; the 0-D (apex) cuboid contains the total number of transactions in the task-relevant data.

Due to the ever-increasing use of data warehouse and OLAP technology, it is possible that a data cube containing the dimensions that are of interest to the user may already exist, fully or partially materialized. If this is the case, we can simply fetch the corresponding aggregate values or compute them using lower-level materialized aggregates, and return the rules needed using a rule generation algorithm. Notice that even in this case, the Apriori property can still be used to prune the search space. If a given k-predicate set has support *sup*, which does not satisfy minimum support, then further exploration of this set should be terminated. This is because any more-specialized version of the k-itemset will have support no greater than *sup* and, therefore, will not satisfy minimum support either. In cases where no relevant data cube exists for the mining task, we must create one on-the-fly. This becomes an iceberg cube computation problem, where the minimum support threshold is taken as the iceberg condition (Chapter 5).

Mining Clustering-Based Quantitative Associations

Besides using discretization-based or data cube–based data sets to generate quantitative association rules, we can also generate *quantitative association rules*

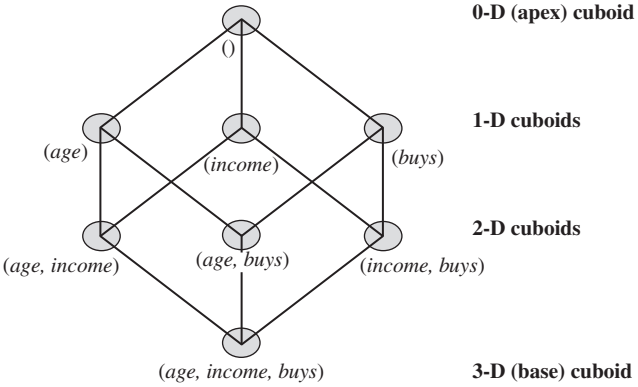

Figure 6.4: Lattice of cuboids, making up a 3-D data cube. Each cuboid represents a different group-by. The base cuboid contains the three predicates *age, income*, and *buys*.

by clustering data in the quantitative dimensions. (Recall that objects within a cluster are similar to one another and dissimilar to those in other clusters.) The general assumption is that interesting frequent patterns or association rules are in general found at relatively dense clusters of quantitative attributes. Here, we describe a top-down approach and a bottom-up approach to clustering that finds quantitative associations.

A typical top-down approach for finding clustering-based quantitative frequent patterns is as follows. For each quantitative dimension, a standard clustering algorithm (e.g., k-means or a density-based clustering algorithm, as described in Chapter 10) can be applied to find clusters in this dimension that satisfy the minimum support threshold. For each cluster, we then examine the 2-D spaces generated by combining the cluster with a cluster or nominal value of another dimension to see if such a combination passes the minimum support threshold. If it does, we continue to search for clusters in this 2-D region and progress to even higher-dimensional combinations. The Apriori pruning still applies in this process: If, at any point, the support of a combination does not have minimum support, its further partitioning or combination with other dimensions cannot have minimum support either.

A bottom-up approach for finding clustering-based frequent patterns works by first clustering in high-dimensional space to form clusters with support that satisfies the minimum support threshold, and then projecting and merging those clusters in the space containing fewer dimensional combinations. However, for high-dimensional data sets, finding high-dimensional clustering itself is a tough problem. Thus, this approach is less realistic.

Using Statistical Theory to Disclose Exceptional Behavior

It is possible to discover quantitative association rules that disclose exceptional behavior, where "exceptional" is defined based on a statistical theory. For example, the following association rule may indicate exceptional behavior:

$$gender = female \Rightarrow meanwage = \$7.90/hr \ (overall\_mean\_wage = \$9.02/hr). \tag{6.6}$$

This rule states that the average wage for females is only \$7.90/hr. This rule is (subjectively) interesting because it reveals a group of people earning a significantly lower wage than the average wage of \$9.02/hr.

An integral aspect of our definition involves applying statistical tests to confirm the validity of our rules. That is, Rule (6.6) is only accepted if a statistical test (in this case, a Z-test) confirms that with high confidence it can be inferred that the mean wage of the female population is indeed lower than the mean wage of the rest of the population[1].

6.1.4 Mining High-Dimensional Data

Our discussions of mining multi-dimensional patterns in the above two subsections are confined to patterns involving a small number of dimensions. However, some applications may need to mine *high-dimensional data* (i.e., data with hundreds or thousands of dimensions). However, it is not easy to extend the previous multi-dimensional pattern mining methods to mine high-dimensional data because the search spaces of such methods grow exponentially with the number of dimensions.

One interesting direction to handle high-dimensional data is to extend a pattern growth approach by exploring the vertical data format to handle data sets with a large number of *dimensions* (also called *features* or *items*, e.g., genes) but a *small* number of *rows* (also called *transactions* or *tuples*, e.g., samples). This is useful in applications like the analysis of gene expressions in bioinformatics, for example, where we often need to analyze microarray data that contain a *large* number of genes (e.g., 10,000 to 100,000) but only a *small* number of samples (e.g., dozens to hundreds).

Another direction is to develop a new methodology that focuses its mining effort on *colossal patterns*, that is, patterns of rather long length, instead of the *complete set* of patterns. One interesting such method is called *Pattern-Fusion*, which takes leaps in the pattern search space, leading to a good approximation of the complete set of colossal frequent patterns. We briefly outline the idea of pattern-fusion here and refer interested readers to the detailed technical paper.

In some applications (e.g., bioinformatics), a researcher can be more interested in finding *colossal* patterns (e.g., long DNA and protein sequences) than finding small (i.e., short) ones since colossal patterns usually carry more significant meanings. Finding colossal patterns is challenging because incremental

[1]The above rule was mined from a real database based on a 1985 U.S. census.

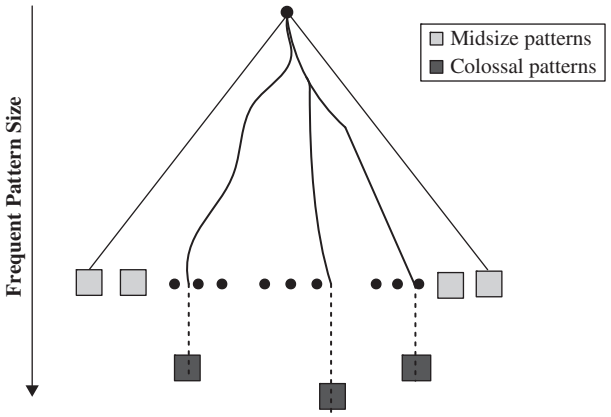

Figure 6.5: A high-dimensional dataset may contain a small set of colossal patterns but exponentially many midsize patterns.

mining tends to get "trapped" by an explosive number of midsize patterns before it can even reach candidate patterns of large size.

All of the pattern mining strategies we have studied so far, such as Apriori and FP-growth, use an incremental growth strategy by nature, that is, they increase the length of candidate patterns by one at a time. Breadth-first search methods like Apriori cannot bypass the generation of an explosive number of midsize patterns generated, making it impossible to reach colossal patterns. Even depth-first search methods like FP-growth can be easily trapped in a huge number of subtrees before reaching colossal patterns. Clearly, a completely new mining methodology is needed to overcome such a hurdle.

As we have observed in Figure 6.5, there could be a small number of colossal patterns (e.g., patterns of size close to 100) but such patterns may generate an exponential number of mid-sized patterns. Instead of mining a complete set of mid-sized patterns, *Pattern-Fusion* fuses a small number of shorter patterns into bigger colossal pattern candidates, and checks against the dataset to see which of such candidates are the true frequent patterns, which can be further fused to generate even larger colossal pattern candidates. Such step-by-step fusing takes leaps in the pattern search space and avoids the pitfalls of both breadth-first and depth-first searches, as shown in Figure 6.6.

Note that a colossal pattern such as $\{a_1, a_2, \ldots, a_{100}\} : 55$ will contain many, many short sub-patterns like $\{a_1, a_2, a_9, \ldots, a_{30}\} : 55$; $\ldots$, $\{a_1, a_9, \ldots, a_{40}\} : 55$; $\ldots$). That is, a colossal pattern should generate far more small patterns than smaller patterns do. Thus, a colossal pattern is more robust in the sense that *if a small number of items are removed from the pattern, the resulting pattern would have a similar support set.* The larger the pattern size, the more prominent this robustness. Such a robustness relationship between a colossal pattern and its corresponding short patterns can be extended to multiple levels.

Thus Pattern-Fusion has the capability to identify good merging candidates,

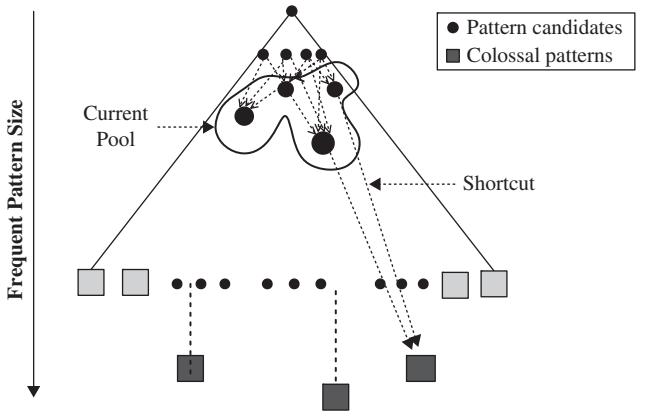

Figure 6.6: Pattern tree traversal: Candidates are taken from a pool of patterns, which results in shortcuts through pattern space to the colossal patterns.

which are the patterns that share some sub-patterns and have some similar support sets. This does help the search leaps through pattern space more directly toward colossal patterns.

It has been theoretically shown that Pattern-Fusion leads to a good approximation of colossal patterns. The method was tested on synthetic and real data sets constructed from program tracing data and microarray data. Experiments show that the method can find most of the colossal patterns with high efficiency.

6.1.5 Mining Rare Patterns and Negative Patterns

All the methods presented so far in this chapter have been for mining frequent patterns. Sometimes, however, it is interesting to find patterns that are rare instead of frequent, or patterns that reflect a negative correlation between items. These patterns are respectively referred to as rare patterns and negative patterns. In this subsection, we consider various ways of defining rare patterns and negative patterns, which are also useful to mine.

Example 6.1.1 **Rare patterns and negative patterns.** In jewelry sales data, sales of diamond watches are rare; however, patterns involving the selling of diamond watches could be interesting. In supermarket data, if we find that customers frequently buy Coca-Cola Classic or Diet Coke but not both, then buying Coca-Cola Classic and buying Diet Coke together is considered a negative (correlated) pattern. In car sales data, a dealer sells a few fuel-thirsty vehicles (e.g., SUVs) to a given customer, and then later sells electric cars to the same customer. Even though buying SUVs and buying electric cars may be negatively correlated events, it can be interesting to discover and examine such exceptional cases.

An **infrequent** (or **rare**) **pattern** is a pattern with a frequency support that is *below* (or *far below*) a user-specified minimum support threshold. However,

since the occurrence frequencies of the majority of itemsets are usually below or even far below the minimum support threshold, it is desirable in practice for users to specify other conditions for rare patterns. For example, if we want to find patterns containing at least one item with a value that is over $500, we should specify such a constraint explicitly. Efficient mining of such itemsets is discussed under mining multidimensional associations (Section 6.1.1), where the strategy is to adopt multiple (e.g., item- or group-based) minimum support thresholds. Other applicable methods are discussed under constraint-based pattern mining (Section 6.3), where user-specified constraints are pushed deep into the iterative mining process.

There are various ways we could define a negative pattern. We will consider three such definitions.

Definition 1 *If itemsets X and Y are both frequent but rarely occur together (i.e., $sup(X \cup Y) < sup(X) \times sup(Y)$), then itemsets X and Y are* ***negatively correlated****, and the pattern $X \cup Y$ is a* ***negatively correlated pattern****. If $sup(X \cup Y) \ll sup(X) \times sup(Y)$, then X and Y are* ***strongly negatively correlated****, and the pattern $X \cup Y$ is a* ***strongly negatively correlated pattern****.*

This definition can easily be extended for patterns containing k-itemsets for $k > 2$.

A problem with the definition, however, is that it is not *null-invariant*. That is, its value can be misleadingly influenced by null transactions, where a *null-transaction* is a transaction that does not contain any of the itemsets being examined (Section 6.3.3). This is illustrated in Example 6.1.2.

Example 6.1.2 **Null-transaction problem with Definition 1.** If there are a lot of null-transactions in the data set, then the number of null-transactions rather than the patterns observed may strongly influence a measure's assessment as to whether a pattern is negatively correlated. For example, suppose a sewing store sells needle packages A and B. The store sold 100 packages each of A and B, but only one transaction contains both A and B. Intuitively, A is negatively correlated with B since the purchase of one does not seem to encourage the purchase of the other.

Let's see how the above definition handles this scenario. If there are 200 transactions, we have $sup(A \cup B) = 1/200 = 0.005$ and $sup(A) \times sup(B) = 100/200 \times 100/200 = 0.25$. Thus, $sup(A \cup B) \ll sup(A) \times sup(B)$, and so Definition 1 indicates that A and B are strongly negatively correlated. What if, instead of only 200 transactions in the database, there are 10^6? In this case, there are many null-transactions, that is, many contain neither A nor B. How does the definition hold up? It computes $sup(A \cup B) = 1/10^6$ and $sup(X) \times sup(Y) = 100/10^6 \times 100/10^6 = 1/10^8$. Thus, $sup(A \cup B) \gg sup(X) \times sup(Y)$, which contradicts the earlier finding even though the number of occurrences of A and B has not changed. The measure in Definition 1 is not null-invariant, where *null-invariance* is essential for quality interestingness measures as discussed in Section 6.3.3.

Definition 2 *If X and Y are strongly negatively correlated, then*

$$sup(X \cup \overline{Y}) \times sup(\overline{X} \cup Y) \gg sup(X \cup Y) \times sup(\overline{X} \cup \overline{Y}).$$

Example 6.1.3 **Null-transaction problem with Definition 2.** Given our needle package example, when there are in total 200 transactions in the database, we have

$$sup(A \cup \overline{B}) \times sup(\overline{A} \cup B) = 99/200 \times 99/200 = 0.245$$
$$\gg sup(A \cup B) \times sup(\overline{A} \cup \overline{B}) = 1/200 \times (200 - 199)/200 \approx 0.25 \times 10^{-4}$$

which, according to Definition 2, indicates that A and B are strongly negatively correlated. However, if there are 10^6 transactions in the database, the measure would compute

$$sup(A \cup \overline{B}) \times sup(\overline{A} \cup B) = 99/10^6 \times 99/10^6 = 9.8 \times 10^{-9}$$
$$\ll sup(A \cup B) \times sup(\overline{A} \cup \overline{B}) = 1/10^6 \times (10^6 - 199)/10^6 \approx 10^{-6}.$$

This time, the measure indicates that A and B are positively correlated, hence, a contradiction. The measure is not null-invariant.

As a third alternative, consider Definition 3, which is based on the Kulczynski measure (i.e., the average of conditional probabilities). It follows the spirit of interestingness measures introduced in Section 6.3.3.

Definition 3 *Suppose that itemsets X and Y are both frequent, that is, $sup(X) \geq min\_sup$ and $sup(Y) \geq min\_sup$, where min_sup is the minimum support threshold. If $(P(X|Y) + P(Y|X))/2 < \epsilon$, where ϵ is a negative pattern threshold, then pattern $X \cup Y$ is a* ***negatively correlated pattern****.*

Example 6.1.4 **Negatively correlated patterns using Definition 3, based on the Kulczynski measure.** Let's reexamine our needle package example. Let min_sup be 0.01% and $\epsilon = 0.02$. When there are 200 transactions in the database, we have $sup(A) = sup(B) = 100/200 = 0.5 > 0.01\%$ and $(P(B|A) + P(A|B))/2 = (0.01 + 0.01)/2 < 0.02$; thus A and B are negatively correlated. Does this still hold true if we have many more transactions? When there are 10^6 transactions in the database, the measure computes $sup(A) = sup(B) = 100/10^6 = 0.01\% \geq 0.01\%$ and $(P(B|A) + P(A|B))/2 = (0.01 + 0.01)/2 < 0.02$, again indicating that A and B are negatively correlated. This matches our intuition. The measure does not have the null-invariance problem of the first two definitions considered.

Let's examine another case: Suppose that among 100,000 transactions, the store sold 1000 needle packages of A but only 10 packages of B; however, every time package B is sold, package A is also sold (i.e., they appear in the same transaction). In this case, the measure computes $(P(B|A) + P(A|B))/2 = (0.01 + 1)/2 = 0.505 \gg 0.02$, which indicates that A and B are positively correlated instead of negatively correlated. This also matches our intuition.

With this new definition of negative correlation, efficient methods can easily be derived for mining negative patterns in large databases. This is left as an exercise for interested readers.

6.2 Mining Compressed or Approximate Patterns

A major challenge in frequent pattern mining is the huge number of discovered patterns. Using a minimum support threshold to control the number of patterns found has limited effect. Too low a value can lead to the generation of an explosive number of output patterns, while too high a value can lead to the discovery of only commonsense patterns.

To reduce the huge set of frequent patterns generated in mining while maintaining high-quality patterns, we can instead mine a compressed or approximate set of frequent patterns. *Top-k most frequent patterns* were proposed to make the mining process concentrate on only the set of k most frequent patterns. Although interesting, they usually do not epitomize the k most representative patterns because of the uneven frequency distribution among itemsets. *Constraint-based mining* of frequent patterns (Section 6.3) incorporates user-specified constraints to filter out uninteresting patterns. Measures of pattern/rule *interestingness* and *correlation* (Section 5.3) can also be used to help confine the search to patterns/rules of interest.

Recall in the last chapter, we introduced two preliminary forms of "compression" of frequent patterns: *closed pattern*, which is a lossless compression of the set of frequent patterns, and *max-pattern*, which is a lossy compression. In this section, we examine two advanced forms of "compression" of frequent patterns that build on the concepts of closed patterns and max-patterns. Section 6.2.1 explores *clustering-based compression of frequent patterns*, which groups patterns together based on their similarity and frequency support. Section 6.2.2 takes a "*summarization" approach*, where the aim is to derive redundancy-aware top-k representative patterns that cover the whole set of (closed) frequent itemsets. The approach considers not only the representativeness of patterns but also their mutual independence to avoid redundancy in the set of generated patterns. The k representatives provide compact compression over the collection of frequent patterns, making them easier to interpret and use.

6.2.1 Mining Compressed Patterns by Pattern Clustering

Pattern compression can be achieved by pattern clustering. Clustering techniques are described in detail in Chapters 9 and 10. In this section, it is not necessary to know the fine details of clustering. Rather, you will learn how the concept of clustering can be applied to compress frequent patterns. Clustering is the automatic process of grouping like objects together, so that objects within a cluster are similar to one another and dissimilar to objects in other clusters. In this case, the objects are frequent patterns. The frequent patterns are clustered using a tightness measure called δ-cluster. A representative pattern is selected for each cluster, thereby offering a compressed version of the set of frequent patterns.

Before we begin, let's review some definitions. An itemset X is a **closed**

frequent itemset in a data set D if X is frequent and there exists no proper super-itemset Y of X such that Y has the same support count as X in D. An itemset X is a **maximal frequent itemset** in data set D if X is frequent and there exists no super-itemset Y such that $X \subset Y$ and Y is frequent in D. Using these concepts alone is not enough to obtain a good representative compression of a data set, as we see in Example 6.2.1.

Example 6.2.1 **Shortcomings of closed itemsets and maximal itemsets for compression**. Table 6.2 shows a subset of frequent itemsets on a large data set, where *a, b, c, d, e, f* represent individual items. There is no non-closed itemset here; therefore, we cannot use closed frequent itemsets to compress the data. The only maximal frequent itemset is P_3. However, we observe that itemsets P_2, P_3, and P_4 are significantly different with respect to their support counts. If we were to use P_3 to represent a compressed version of the data, we would lose this support count information entirely. Consider the two pairs (P_1, P_2) and (P_4, P_5). From visual inspection, the patterns within each pair are very similar with respect to their support and expression. Therefore, intuitively, P_2, P_3, and P_4, collectively, should serve as a better compressed version of the data.

Table 6.2: Subset of Frequent Itemsets

ID	*Itemsets*	*Support*
P_1	$\{b, c, d, e\}$	205,227
P_2	$\{b, c, d, e, f\}$	205,211
P_3	$\{a, b, c, d, e, f\}$	101,758
P_4	$\{a, c, d, e, f\}$	161,563
P_5	$\{a, c, d, e\}$	161,576

Let's see if we can find a way of clustering frequent patterns as a means of obtaining a compressed representation of them. We will need to define a good similarity measure, cluster patterns according to this measure, and then select and output only a *representative pattern* for each cluster. Since the set of closed frequent patterns is a lossless compression over the original frequent patterns set, it is a good idea to discover representative patterns around the collection of *approximately closed* patterns.

We can use the following distance measure between closed patterns. Let P_1 and P_2 be two closed patterns. Their supporting transaction sets are $T(P_1)$ and $T(P_2)$, respectively. The **pattern distance** of P_1 and P_2, $Pat\_Dist(P_1, P_2)$, is defined as

$$Pat\_Dist(P_1, P_2) = 1 - \frac{|T(P_1) \cap T(P_2)|}{|T(P_1) \cup T(P_2)|}. \tag{6.7}$$

Pattern distance is a valid distance metric defined on the set of transactions. Note that it incorporates the *support* information of patterns, as desired previ-

ously.

Example 6.2.2 **Pattern distance.** Suppose P_1 and P_2 are two patterns such that $T(P_1) = \{t_1, t_2, t_3, t_4, t_5\}$ and $T(P_2) = \{t_1, t_2, t_3, t_4, t_6\}$, where t_i is a transaction in the database. The distance between P_1 and P_2 is $Pat\_Dist(P_1, P_2) = 1 - \frac{4}{6} = \frac{1}{3}$.

Now, let's consider the *expression* of patterns. Given two patterns A and B, we say B can be **expressed** by A if $O(B) \subset O(A)$, where $O(A)$ is the corresponding itemset of pattern A. Following this definition, assume patterns $P_1, P_2, \ldots, P_k$ are in the same cluster. The representative pattern P_r of the cluster should be able to *express* all the other patterns in the cluster. Clearly, we have $\cup_{i=1}^{k} O(P_i) \subseteq O(P_r)$.

Using the distance measure, we can simply apply a clustering method, such as k-means (Section 9.2), on the collection of frequent patterns. However, this introduces two problems. First, the quality of the clusters cannot be guaranteed; second, it may not be able to find a representative pattern for each cluster (i.e., the pattern P_r may not belong to the same cluster). To overcome these problems, this is where the concept of δ-cluster comes in, where δ $(0 \leq \delta \leq 1)$ measures the tightness of a cluster.

A pattern P is **δ-covered** by another pattern P' if $O(P) \subseteq O(P')$ and $Pat\_Dist(P, P') \leq \delta$. A set of patterns form a **δ-cluster** if there exists a representative pattern P_r such that for each pattern P in the set, P is δ-covered by P_r.

Note that according to the concept of δ-cluster, a pattern can belong to multiple clusters. Also, using δ-cluster, we only need to compute the distance between each pattern and the representative pattern of the cluster. Because a pattern P is δ-covered by a representative pattern P_r only if $O(P) \subseteq O(P_r)$, we can simplify the distance calculation by considering only the supports of the patterns:

$$Pat\_Dist(P, P_r) = 1 - \frac{|T(P) \cap T(P_r)|}{|T(P) \cup T(P_r)|} = 1 - \frac{|T(P_r)|}{|T(P)|}. \tag{6.8}$$

If we restrict the representative pattern to be frequent, then the number of representative patterns (i.e., clusters) is no less than the number of maximal frequent patterns. This is because a maximal frequent pattern can only be covered by itself. To achieve more succinct compression, we relax the constraints on representative patterns, that is, we allow the support of representative patterns to be *somewhat* less than *min_sup*.

For any representative pattern P_r, assume its support is k. Since it has to *cover* at least one frequent pattern (i.e., P) with support that is at least *min_sup*, we have

$$\delta \geq Pat\_Dist(P, P_r) = 1 - \frac{|T(P_r)|}{|T(P)|} \geq 1 - \frac{k}{min\_sup}. \tag{6.9}$$

That is, $k \geq (1 - \delta) \times min\_sup$. This is the minimum support for a representative pattern, denoted as min_sup_r.

Based on the preceding discussion, the pattern compression problem can be defined as follows: *Given a transaction database, a minimum support* min_sup, *and the cluster quality measure δ, the pattern compression problem is to find a set of representative patterns R such that for each frequent pattern P (with respect to* min_sup), *there is a representative pattern $P_r \in R$ (with respect to* min_sup_r), *which covers P, and the value of $|R|$ is minimized.*

Finding a minimum set of representative patterns is an NP-Hard problem. However, efficient methods have been developed that reduce the number of closed frequent patterns generated by orders of magnitude with respect to the original collection of closed patterns. The methods succeed in finding a high-quality compression of the pattern set.

6.2.2 Extracting Redundancy-Aware Top-k Patterns

Mining the top-k most frequent patterns is a strategy for reducing the number of patterns returned during mining. However, in many cases, frequent patterns are not mutually independent but often clustered in small regions. This is somewhat like finding 20 population centers in the world, which may result in cities clustered in a small number of countries rather than evenly distributed across the globe. Instead, most users would prefer to derive the k most interesting patterns, which are not only significant, but also mutually independent and containing little redundancy. A small set of k representative patterns that have not only high significance but also low redundancy are called **redundancy-aware top-k patterns**.

Example 6.2.1 **Redundancy-aware top-k strategy versus other top-k strategies.** Figure 6.7 illustrates the intuition behind *redundancy-aware top-k patterns* versus *traditional top-k patterns* and *k-summarized patterns*. Suppose we have the frequent patterns set shown in Figure 6.7(a), where each circle represents a pattern of which the significance is colored in grayscale. The distance between two circles reflects the redundancy of the two corresponding patterns: The closer the circles are, the more redundant the respective patterns are to one another. Let's say we want to find three patterns that will best represent the given set, that is, $k = 3$. Which three should we choose?

Arrows are used to show the patterns chosen if using redundancy-aware top-k patterns (Figure 6.7b), traditional top-k patterns (Figure 6.7c), or k-summarized patterns (Figure 6.7d). In Figure 6.7(c), the **traditional top-k strategy** relies solely on significance: It selects the three most significant patterns to represent the set.

In Figure 6.7(d), the ***k*-summarized pattern strategy** selects patterns based solely on nonredundancy. It detects three clusters, and finds the most representative patterns to be the "centermost'" pattern from each cluster. These patterns are chosen to represent the data. The selected patterns are considered "summarized patterns" in the sense that they represent or "provide a summary" of the clusters they stand for.

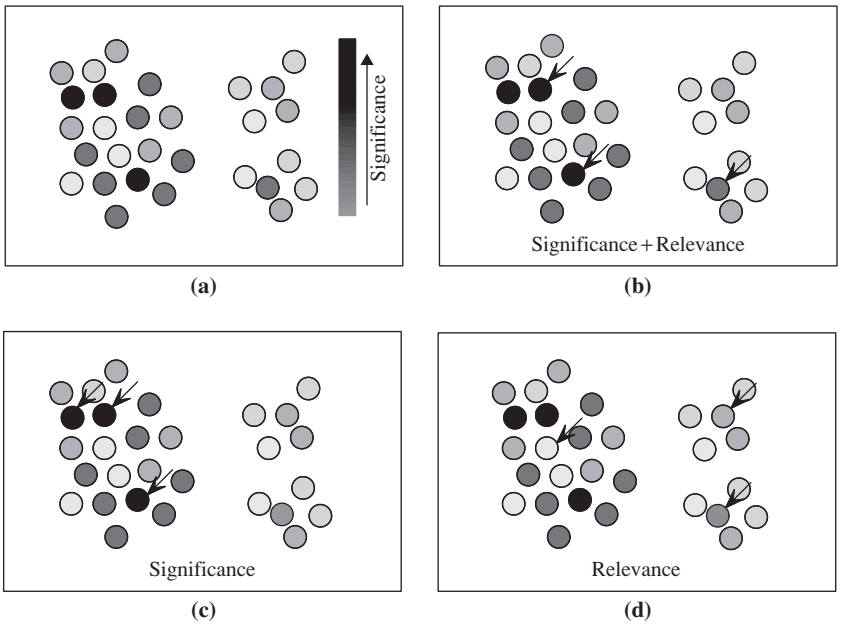

Figure 6.7: Conceptual view comparing top-k methodologies (where gray levels represent pattern significance, and the closer that two patterns are displayed, the more redundant they are to one another): (a) original patterns, (b) redundancy-aware top-k patterns, (c) traditional top-k patterns, and (d) k-summarized patterns.

By contrast, in Figure 6.7(b) the **redundancy-aware top-k patterns** make a trade-off between significance and redundancy. The three patterns chosen here have high significance and low redundancy. Observe, for example, the two highly significant patterns that, based on their redundancy, are displayed next to each other. The redundancy-aware top-k strategy selects only one of them, taking into consideration that two would be redundant. To formalize the definition of redundancy-aware top-k patterns, we'll need to define the concepts of significance and redundancy.

A **significance measure** S is a function mapping a pattern $p \in \mathcal{P}$ to a real value such that $S(p)$ is the degree of interestingness (or usefulness) of the pattern p. In general, significance measures can be either objective or subjective. *Objective measures* depend only on the structure of the given pattern and the underlying data used in the discovery process. Commonly used objective measures include support, confidence, correlation, and *tf-idf* (or *term frequency* versus *inverse document frequency*), where the latter is often used in information retrieval. *Subjective* measures are based on user beliefs in the data. They therefore depend on the users who examine the patterns. A subjective measure is usually a relative score based on user prior knowledge or a background model. It often measures the unexpectedness of a pattern by computing its divergence

from the background model. Let $S(p, q)$ be the **combined significance** of patterns p and q, and $S(p|q) = S(p, q) - S(q)$ be the **relative significance** of p given q. Note that the combined significance, $S(p, q)$, means the collective significance of two individual patterns p and q, not the significance of a single super pattern $p \cup q$.

Given the significance measure S, the **redundancy R between two patterns** p and q is defined as $R(p, q) = S(p) + S(q) - S(p, q)$. Subsequently, we have $S(p|q) = S(p) - R(p, q)$.

We assume that the combined significance of two patterns is no less than the significance of any individual pattern (since it is a collective significance of two patterns) and does not exceed the sum of two individual significance patterns (since there exists redundancy). That is, the redundancy between two patterns should satisfy

$$0 \leq R(p, q) \leq \min(S(p), S(q)). \tag{6.10}$$

The ideal redundancy measure $R(p, q)$ is usually hard to obtain. However, we can approximate redundancy using distance between patterns such as with the distance measure defined in Section 6.2.1.

The problem of finding redundancy-aware top-k patterns can thus be transformed into finding a k-pattern set that maximizes the marginal significance, which is a well-studied problem in information retrieval. In this field, a document has high marginal relevance if it is both relevant to the query and contains minimal marginal similarity to previously selected documents, where the marginal similarity is computed by choosing the most relevant selected document. Experimental studies have shown this method to be efficient and able to find high-significance and low-redundancy top-k patterns.

6.3 Constraint-Based Pattern Mining

A pattern mining process may uncover thousands of patterns from a given data set, many of which may end up being unrelated or uninteresting to users. Often, a user have a good sense of which "direction" of mining may lead to interesting patterns and the "form" of the patterns or rules they want to find. They may also have a sense of "conditions" for the rules, which would eliminate the discovery of certain rules that they know would not be of interest. Thus, a good option is to have users specify such intuition or expectations as *constraints* to confine the search space or perform constraint refinement interactively based on the intermediate mining results. This strategy is known as **constraint-based mining**. The constraints can include the following:

- **Knowledge type constraints:** These specify the type of knowledge to be mined, such as association, correlation, classification, or clustering.
- **Data constraints:** These specify the set of task-relevant data.
- **Dimension/level constraints:** These specify the desired dimensions (or attributes) of the data, the abstraction levels, or the level of the concept hierarchies to be used in mining.
- **Interestingness constraints:** These specify thresholds on statistical measures of rule interestingness such as support, confidence, and correlation.
- **Rule/pattern constraints:** These specify the form of, or conditions on, the rules/patterns to be mined. Such constraints may be expressed as metarules (rule templates), as the maximum or minimum number of predicates that can occur in the rule antecedent or consequent, or as relationships among attributes, attribute values, and/or aggregates.

These constraints can be specified using a high-level data mining query language or a template-based graphical user interface.

The first four constraint types have already been addressed earlier in the book. In this section, we discuss the use of *rule/pattern constraints* to focus on the mining task. This form of constraint-based mining allows users to describe the rules or patterns that they would like to uncover, thereby making the data mining process more *effective*. In the meantime, a sophisticated mining query optimizer can be used to exploit the constraints specified by the user, thereby making the mining process more *efficient*.

In some cases, a user may like to specify some syntactic form of rules (also called *meta-rules*) that she is interested in mining. Such syntactic forms help the user to express her expectation and also help the system to confine search space and improve mining efficiency.

For example, a meta-rule can be in the form of

$$P_1(X, Y) \wedge P_2(X, W) \Rightarrow \mathit{buys}(X, \text{“}\mathit{iPad}\text{”}), \tag{6.11}$$

where P_1 and P_2 are **predicate variables** which can be instantiated to attributes in a given database during the mining process, X is a variable representing a customer, and Y and W take on values of the attributes assigned to P_1 and P_2, respectively. Typically, a user can specify a list of attributes to be considered for instantiation with P_1 and P_2. Otherwise, a default set may be used.

A metarule forms a hypothesis regarding the relationships that the user is interested in probing or confirming. Following such a template, a system can then mine concrete rules that match the given metarule. Possibly, Rule (6.12) that complies with Metarule (6.11) will be returned as mining results

$$age(X, \text{“20..29”}) \wedge income(X, \text{“41}K..60K\text{”}) \Rightarrow buys(X, \text{“}iPad\text{”}). \quad (6.12)$$

In order to generate interesting and useful mining results, users may have multiple ways to specify rule/pattern constraints. It is desirable for a mining system to use rule/pattern constraints to prune the search space, that is, to push such constraints deeply into the mining process while still ensure the completeness of the answer returned for a mining query. However, this is a nontrivial task, and its study leads to *constraint-based pattern mining*.

To study how to use constraints at mining frequent patterns or association rules, we examine the following running example.

Example 6.3.1 **Constraints on shopping transaction mining.** Suppose that a multidimensional shopping transaction database contains the following interrelated relations:

- *item(item_ID, item_name, description, category, price)*
- *sales(transaction_ID, day, month, year, store_ID, city)*
- *trans_item(item_ID, transaction_ID)*

Here, the *item* table contains attributes *item_ID, item_name, description, category*, and *price*; the *sales* table contains attributes *transaction_ID, day, month, year, store_ID*, and *city*; and the two tables are linked via the foreign key attributes, *item_ID* and *transaction_ID*, in the table *trans_item*.

A mining query may contain multiple constraints, For example, we may have a query: “*From the sales in Chicago in 2020, find the patterns (i.e., item sets) that which cheap items (where the sum of the prices is less than $10) appear in the same transaction with (hence may promote) which expensive items (where the minimum price is $50).*”

This query contains the following four constraints: (1) $sum(I.price) < \$10$, where I represents the *item_ID* of a cheap item; (2) $min(J.price) \geq \$50$), where J represents the *item_ID* of an expensive item; (3) $T.city = Chicago$; and (4) $T.year = 2020$, where T represents a *transaction_ID*.

In constraint-based pattern mining, the search space can be pruned in the mining process with two strategies: *pruning pattern search space* and *pruning*

data search space. The former checks candidate patterns and decides whether a pattern should be eliminated from further processing. For example, it may prune a pattern if all of its superpattern will be useless in the remaining mining process, say, based on the Apriori property. The latter checks the data set to determine whether a particular data object will not be able to contribute to the subsequent generation of satisfiable patterns in the remaining mining process (hence safely pruning the data object).

We examine these pruning strategies in the following subsections.

6.3.1 Pruning Pattern Space with Pattern Pruning Constraints

Based on how a constraint may interact with the pattern mining process, we partition pattern mining constraints into four categories: (1) *antimonotonic,* (2) *monotonic,* (3) *convertible,* and (4) *nonconvertible*. Let's examine them one by one.

Pattern Antimonotonicity

The first group of constraints are characterized with **pattern antimonotonicity**. A constraint C is **pattern antimonotonic** if it has the following property: *If an itemset does not satisfy constraint C, none of its supersets will satisfy C.*

Let's examine a constraint "$C_1 : sum(I.price) \leq \$100$" and see what may happen if the constraint is added to our shopping transaction mining query. Suppose we are mining itemsets of size k at the kth iteration using the Apriori algorithm or the like. If the summation of the prices of the items in a candidate itemset S_1 is greater than \$100, this itemset should be pruned from the search space, since not only the current set cannot satisfy the constraint, but also adding more items into the set (assuming that the price of any item is no less than zero) will never be able to satisfy the constraint. Notice that the pruning of this pattern (frequent itemset) for constraint C_1 is not confined to the Apriori candidate-generation-and-test framework. For example, for the same reason, S_1 should be pruned in the pattern-growth framework since pattern S_1 and the further growth from it can never make constraint C_1 satisfiable.

This property is called *antimonotonicity* because *monotonicity* of a constraint usually means *if a pattern p satisfies a constraint C, its further expansion will always satisfy C*, however, here we claim that this constraint may have a *reverse behavior*: *once a pattern p_1 violates the constraint C_1, its further growth (or expansion) will always violate C_1*. Pattern pruning by antimonotonicity can be applied at each iteration of Apriori-style algorithms to help improve the efficiency of the overall mining process while guaranteeing the completeness of the data mining task.

It is interesting to note that the very basic Apriori property itself (which states that all nonempty subsets of a frequent itemset must also be frequent) is antimonotonic: *If an itemset does not satisfy the minimum support threshold, none of its supersets can.* This property has been used at each iteration of the

Apriori algorithm to reduce the number of candidate itemsets to be examined, thereby reducing the search space for frequent pattern mining.

There are many constraints that are antimonotonic. For example, the constraint "$min(J.price) \geq \$50$," and "$count(I) \leq 10$," are antimonotonic. However, there are also many constraints that are not antimonotonic. For example, the constraint "$avg(I.price) \leq \$10$" is not antimonotonic. This is because even for a given itemset S that does not satisfy this constraint, a superset created by adding some (cheap) items may make it satisfy the constraint. Hence, pushing this constraint inside the mining process will not guarantee the completeness of the data mining process. A list of popularly encountered constraints is given in the first column of Table 6.3. The antimonotonicity of the constraints is indicated in the second column. To simplify our discussion, only existence operators (e.g., $=$, $\in$, but not $\neq$, $\notin$) and comparison (or containment) operators with equality (e.g., $\leq$, $\subseteq$) are given.

Pattern Monotonicity

The second category of constraints is **pattern monotonicity**. A constraint C is **pattern monotonic** if it has the following property: *If an itemset satisfies constraint C, all of its supersets will satisfy C.*

Let's examine another constraint "$C_2 : sum(I.price) \geq \$100$," and see what may happen if the constraint is added to our example query. Suppose we are mining itemsets of size k at the kth iteration using the Apriori algorithm or the like. If the summation of the prices of the items in a candidate itemset S_1 is less than \$100, this itemset should not be pruned from the search space, since adding more items to the current set may make the itemset satisfy the constraint. However, once the sum of the prices of the items in itemset S satisfies the constraint C_2, there is no need to check this constraint for S any more since adding more items will not decrease the sum value and will always satisfy the constraint. In other words, if an itemset satisfies the constraint, so do all of its supersets. Please note that the property is independent of particular iterative pattern mining algorithms. For example, the same pruning methodology should be adopted for pattern-growth algorithms as well.

There are many pattern monotonic constraints in practice. For example, "$min(I.price) \leq \$10$," and "$count(I) \geq 10$," are such constraints. The pattern monotonicity of the list of frequently encountered constraints is indicated in the third column of Table 6.3.

Convertible Constraints: Ordering Data in Transactions

There are constraints that are neither pattern antimonotonic nor pattern monotonic. For example, it is hard to directly push the constraint "$C_3 : avg(I.price) \leq \$10$" deeply into an iterative mining process because the next item to be added to the current itemset can be more expensive or less expensive than the average price of the itemset computed so far. At the first glance, it seems to be hard to explore constraint pushing for such kind of constraints in pattern mining.

Table 6.3: Characterization of Commonly Used Pattern Pruning Constraints

Constraint	*Antimonotonic*	*Monotonic*	*Succinct*
$v \in S$	no	yes	yes
$S \supseteq V$	no	yes	yes
$S \subseteq V$	yes	no	yes
$min(S) \leq v$	no	yes	yes
$min(S) \geq v$	yes	no	yes
$max(S) \leq v$	yes	no	yes
$max(S) \geq v$	no	yes	yes
$count(S) \leq v$	yes	no	no
$count(S) \geq v$	no	yes	no
$sum(S) \leq v$ ($\forall a \in S$, $a \geq 0$)	yes	no	no
$sum(S) \geq v$ ($\forall a \in S$, $a \geq 0$)	no	yes	no
$range(S) \leq v$	yes	no	no
$range(S) \geq v$	no	yes	no
$avg(S)\ \theta\ v$, $\theta \in \{\leq, \geq\}$	convertible	convertible	no
$support(S) \geq \xi$	yes	no	no
$support(S) \leq \xi$	no	yes	no
$all\_confidence(S) \geq \xi$	yes	no	no
$all\_confidence(S) \leq \xi$	no	yes	no

However, observing that the items in a transaction can be treated as a set, and thus it is possible to arrange items in a transaction in any specific ordering. Interestingly, when the items in the itemset are arranged in a price ascending or descending order, it is possible to explore efficient pruning in frequent itemset mining as we did before. In this context, it is possible to convert such kind of constraints into monotonic or antimonotonic constraints. Hence we call such constraints as **convertible constraints**.

Let's re-examine the constraint C_3. If the items in all the transactions are sorted in the price-ascending order (or items in any transaction are added in this order) in the pattern-growth mining process, the constraint C_3 becomes *antimonotonic*, because if an itemset I violates the constraint (i.e., with an average price greater than \$10), then further addition of more expensive items into the itemset will never make it satisfy the constraint. Similarly, if items in all the transaction are sorted (or being added to the itemset being mined) in the price-descending order, it becomes *monotonic,* because if the itemset satisfies the constraint (i.e., with an average price no greater than \$10), then adding cheaper items into the current itemset will still make the average price no greater than \$10.

Will the Apriori-like algorithm make good use of the convertible constraint to prune its search space? Unfortunately, such a constraint satifaction checking cannot be done easily with an Apriori-like candidate-generation-and-test algorithm, because an Apriori-like algorithm requires all of the subsets (say, $\{ab\}$,

$\{bc\}$, $\{ac\}$) of a candidate $\{abc\}$ must be frequent and satisfies the constraint. However, even $\{abc\}$ itself could be a valid itemset (i.e., $avg(\{abc\}.price) \leq \$10$), the subset $\{bc\}$ may have violated C_3 and we will never be able to generate $\{abc\}$ since $\{bc\}$ has been pruned.

Let S represent a set of items and its value be *price*. Besides "$avg(S) \leq c$" and "$avg(S) \geq c$", there are also other convertible constraints. For example, "$variance(S) \geq c$", "$standard\_deviation(S) \geq c$" are convertible constraints. But this does not imply that every non-monotonic or non-antimonotonic constraint is convertible. For example, if the aggregation function for item values in the set has random sampling behavior, it will be hard to arrange the items in a monotonically increasing or decreasing order. Therefore, there still exists a category of constraints that are **nonconvertible**. The good news is that although there exist some tough constraints that are not convertible, most simple and frequently used constraints belong to one of the three categories we just described, antimonotonic, monotonic and convertible, to which efficient constraint mining methods can be applied.

6.3.2 Pruning Data Space with Data Pruning Constraints

The second way of search space pruning in constraint-based frequent pattern mining is *pruning data space*. This strategy prunes pieces of data if they will not contribute to the subsequent generation of satisfiable patterns in the mining process. We examine *data antimonotonicity* in this section.

Interestingly, many constraints are **data-antimonotonic** in the sense that *during the mining process*, if a data entry cannot satisfy a data-antimonotonic constraint based on the current pattern, then it can be pruned. We prune it because it will not be able to contribute to the generation of any superpattern of the current pattern in the remaining mining process.

Example 6.3.1 **Data antimonotonicity.** We examine constraint C_1 : $sum(I.price) \geq \$100$, that is, the sum of the prices of the items in the mined pattern must be no less than \$100. Suppose that the current frequent itemset, S, does not satisfy constraint C_1 (say, because the sum of the prices of the items in S is \$50). If the remaining frequent items in a transaction T_i cannot make S satisfy the constraint (e.g., the remaining frequent items in T_i are $\{i_2.price = \$5, i_5.price = \$10, i_8.price = \$20\}$), then T_i cannot contribute to the patterns to be mined from S, and can be pruned from further mining.

Note that such pruning may not be effective by enforcing it only at the beginning of the mining process. This is because although it may prune those transations whose sum of items do not satisfy the constraint C_1. However, we may encounter a case that $i_3.price = \$90$, but later in the mining process, i_3 becomes infrequent with S in the transaction data set, and at this point, T_i should be be pruned. Therefore, such checking and pruning should be enforced at each iteration to reduce the data search space.

Notice that constraint C_1 is a monotonic constraint with respect to pattern space pruning. As we have seen, this pattern monotonic constraint has very

limited power for reducing the search space in pattern pruning. However, the same constraint is data antimonotonic and can be used for effective reduction of the data search space.

For a pattern antimonotonic constraint, such as $C_2 : sum(I.price) \leq \$100$, we can prune both pattern and data search spaces at the same time. Based on our study of pattern pruning, we already know that the current itemset can be pruned if the sum of the prices in it is over \$100 (since its further expansion can never satisfy C_2). At the same time, we can also prune any remaining items in a transaction T_i that cannot make the constraint C_2 valid. For example, if the sum of the prices of items in the current itemset S is \$90, any item with price over \$10 in the remaining frequent items in T_i can be pruned. If none of the remaining items in T_i can make the constraint valid, the entire transaction T_i should be pruned.

Consider pattern constraints that are neither antimonotonic nor monotonic such as "$C_3 : avg(I.price) \leq 10$." These can be data-antimonotonic because if the remaining items in a transaction T_i cannot make the constraint valid, then T_i can be pruned as well. Therefore, data-antimonotonic constraints can be quite useful for constraint-based data space pruning.

Notice that search space pruning by data antimonotonicity is confined only to a pattern growth-based mining algorithm because the pruning of a data entry is determined based on whether it can contribute to a specific pattern. Data antimonotonicity cannot be used for pruning the data space if the Apriori algorithm is used because the data are associated with all of the currently active patterns. At any iteration, there are usually many active patterns. A data entry that cannot contribute to the formation of the superpatterns of a given pattern may still be able to contribute to the superpattern of other active patterns. Thus, the power of data space pruning can be very limited for nonpattern growth-based algorithms.

6.3.3 Mining Space Pruning with Succinctness Constraints

For pattern mining, there is another category of constraints called *succinct constraints*. A constraint c is **succinct** if it can be enforced by *directly pruning some data objects from the database* or by *directly enumerating all and only those sets that are guaranteed to satisfy the constraint*. The former is called **data succinct** since it enables direct data space pruning, whereas the latter is called **pattern succinct** since it enables direct pattern generation by starting with initial patterns that satisfy the constraint. Let's examine a few examples.

First, let's examine the constraint $i \in S$, that is, the pattern must contain item i. To find the patterns containing item i, one can mine only i-projected database since a transaction does not contain i will not contribute to the patterns containing i, and for those containing i, all the remaining items can participate the remaining of the mining process. This facilitates data space pruning at the beginning and thus this constraint is both data and pattern succinct. On the other hand, to find the patterns that do not contain item i (i.e., $i \notin S$), one can mine it by mining the transaction database with i removed since i in a

transaction will not contribute to the pattern. This facilitates data space pruning at the beginning and also facilitate pattern space pruning (since it avoids mine any intermediate patterns containing i, thus the constraint is succinct and it is both pattern succinct and data succinct.

As another example, a constraint "$min(S.price) \geq \$50$" is data succinct since we can remove all items whose price is less than \$50 from the transactions since any item whose price is no less than \$50 will not contribute to the pattern mining process. Similarly, $min(S.Price) \leq v$ is pattern succinct since we can start with only those items whose price is no greater than v.

Notice not all the constraints are succinct. For example, the constraint $sum(S.Price) \geq v$ is not succinct because it cannot be used to facilitate the pruning of any item from a transaction at the beginning of the process since the sum of the price of an itemset S will keep increasing.

The pattern succinctness of the list of SQL primitives–based constraints is indicated in the fourth column of Table 6.3.

From the above discussion, we can see that the same constraint may belong to more than one category. For example, the constraint "$min(I.price) \leq \$10$" is pattern monotonic and also data succinct. In this case, we can use data succinctness to start only with those items whose price is no more than \$10. By doing so, it has implicitly applied pattern monotonicity property already since once the constraint is used at the starting point (i.e., satisfied), we will not need to check it any more. As another example, the constraint "$c_0 : sum(I.price) \geq \$100$" is both pattern monotonic and data anti-monotonic, we can use the data antimonotonicity to prune those transactions whose prices of the remaining items adding together cannot reach \$100. In the meantime, once a pattern satisfies c_0, we will not need to check c_0 again in the mining process.

In applications, a user may pose a mining query which may contain multiple constraints. In many cases, multiple constraints can be enforced together to jointly prune mining space, which may lead to more efficient processing. However, in some cases, different constraints may require different item-ordering for the effective constraint enforcement, especially for convertible constraints. For example, a query may contain both $c_1 : avg(S.profit) > 20$ and $c_2 : avg(S.price) < 50$. Unfortunately, sorting on profit in value-descending order may not result in value-descending order of their associated item price. In this case, it is the best to estimate which ordering may lead to more effective pruning, and mining following the more effective pruning ordering will lead to more efficient processing. Suppose it is hard to find patterns satisfying c_1 but easy to find pattern satisfying c_2. Then the system should sort the items in transactions in profit descending ordering. Once the average profit of the current itemset drops to below \$20, the itemset can be tossed (i.e., no further mining with it), which will lead to efficient processing.

6.4 Mining Sequential Patterns

A **sequence database** consists of sequences of ordered elements or events, recorded with or without a concrete notion of time. There are many applications involving sequence data. Typical examples include customer shopping sequences, Web clickstreams, biological sequences, and sequences of events in science and engineering, and in natural and social developments. In this section, we study *sequential pattern mining* in transactional databases, and with proper extensions, such mining algorithms can help find sequential patterns for many other applications, such as finding sequential patterns for Webclick streams, and for science, engineering and social event mining. We start with the basic concepts of sequential pattern mining in Section 6.4.1. Section 6.4.2 presents several scalable methods for such mining. We will discuss constraint-based sequential pattern mining in Section 6.4.3.

6.4.1 Sequential Pattern Mining: Concepts and Primitives

"What is sequential pattern mining?" **Sequential pattern mining** is the mining of frequently occurring ordered events or subsequences as patterns. An example of a sequential pattern is "*Customers who buy an iPad Pro are likely to buy an Apple pencil within 90 days*". For retail data, sequential patterns are useful for shelf placement and promotions. This industry, as well as telecommunications and other businesses, may also use sequential patterns for targeted marketing, customer retention, and many other tasks. Other areas in which sequential patterns can be applied include Web access pattern analysis, production processes, and network intrusion detection. Notice that most studies of sequential pattern mining concentrate on *categorical* or *symbolic patterns*, whereas numerical curve analysis usually belongs to the scope of trend analysis and forecasting in statistical time-series analysis discussed in many statistics or time-series analysis textbooks.

The sequential pattern mining problem was first introduced by Agrawal and Srikant in 1995 based on their study of customer purchase sequences, as follows: *Given a set of sequences, where each sequence consists of a list of events (or elements) and each event consists of a set of items, and given a user-specified minimum support threshold of min_sup, sequential pattern mining finds all* ***frequent*** *subsequences, that is, the subsequences whose occurrence frequency in the set of sequences is no less than min_sup.*

Let's establish some vocabulary for our discussion of sequential pattern mining. Let $\mathcal{I} = \{I_1, I_2, \ldots, I_p\}$ be the set of all *items*. An **itemset** is a non-empty set of items. A **sequence** is an ordered list of **events**. A sequence s is denoted $\langle e_1 e_2 e_3 \cdots e_l \rangle$, where event e_1 occurs before e_2, which occurs before e_3, and so on. Event e_j is also called an **element** of s. In the case of customer purchase data, an event refers to a shopping trip in which a customer bought items at a certain store. The event is thus an itemset, that is, an unordered list of items that the customer purchased during the trip. The itemset (or event) is denoted

$(x_1 x_2 \cdots x_q)$, where x_k is an item. For brevity, the brackets are omitted if an element has only one item, that is, element (x) is written as x. Suppose that a customer made several shopping trips to the store. These ordered events form a sequence for the customer. That is, the customer first bought the items in s_1, then later bought the items in s_2, and so on. An item can occur at most once in an event of a sequence[2], but can occur multiple times in different events of a sequence. The number of instances of items in a sequence is called the **length** of the sequence. A sequence with length l is called an **l-sequence**. A sequence $\alpha = \langle a_1 a_2 \cdots a_n \rangle$ is called a **subsequence** of another sequence $\beta = \langle b_1 b_2 \cdots b_m \rangle$, and β is a **supersequence** of α, denoted as $\alpha \sqsubseteq \beta$, if there exist integers $1 \leq j_1 < j_2 < \cdots < j_n \leq m$ such that $a_1 \subseteq b_{j_1}$, $a_2 \subseteq b_{j_2}$, $\ldots$, $a_n \subseteq b_{j_n}$. For example, if $\alpha = \langle (ab), d \rangle$ and $\beta = \langle (abc), (de) \rangle$ where a, b, c, d, and e are items, then α is a subsequence of β and β is a supersequence of α.

A **sequence database**, S, is a set of tuples, $\langle SID, s \rangle$, where SID is a *sequence_ID* and s is a sequence. For our example, S contains sequences for all customers of the store. A tuple $\langle SID, s \rangle$ is said to **contain** a sequence α, if α is a subsequence of s. The **support** of a sequence α in a sequence database S is the number of tuples in the database containing α, that is, $support_S(\alpha) = | \{\langle SID, s \rangle | (\langle SID, s \rangle \in S) \wedge (\alpha \sqsubseteq s)\} |$. It can be denoted as $support(\alpha)$ if the sequence database is clear from the context. Given a positive integer min_sup as the **minimum support threshold**, a sequence α is **frequent** in sequence database S if $support_S(\alpha) \geq min\_sup$. That is, for sequence α to be frequent, it must occur at least min_sup times in S. *A frequent sequence is called a* **sequential pattern**. A sequential pattern with length l is called an **l-pattern**. The following example illustrates these concepts.

Example 6.4.1 **Sequential patterns.** Consider the sequence database, S, given in Table 6.4, which will be used in examples throughout this section. Let $min\_sup = 2$. The set of *items* in the database is $\{a, b, c, d, e, f, g\}$. The database contains four sequences.

Sequence_ID	*Sequence*
1	$\langle a(abc)(ac)d(cf) \rangle$
2	$\langle (ad)c(bc)(ae) \rangle$
3	$\langle (ef)(ab)(df)cb \rangle$
4	$\langle eg(af)cbc \rangle$

Table 6.4: A sequence database

Let's have a close look at *sequence* 1, which is $\langle a(abc)(ac)d(cf) \rangle$. It has five *events*, namely (a), (abc), (ac), (d) and (cf), which occur in the order listed. Items a and c each appear more than once in different events of the sequence. There are nine instances of items in sequence 1, therefore it has a *length* of nine and is called a 9-*sequence*. Item a occurs three times in sequence 1 and so

[2]we simplify our discussion here in the same spirit as frequent itemset mining, but the developed method can be extended to consider multiple identical items.

contributes three to the length of the sequence. However, the entire sequence contributes only one to the *support* of $\langle a \rangle$. Sequence $\langle a(bc)df \rangle$ is a *subsequence* of sequence 1 since the events of the former are each subsets of events in sequence 1, and the order of events is preserved. Consider subsequence $s = \langle (ab)c \rangle$. Looking at the sequence database, S, we see that sequences 1 and 3 are the only ones that *contain* the subsequence s. The support of s is thus 2, which satisfies minimum support. Therefore, s is frequent, and so we call it a *sequential pattern*. It is a 3-*pattern* since it is a sequential pattern of length three.

This model of sequential pattern mining is an abstraction of customer-shopping sequence analysis. Scalable methods for sequential pattern mining on such data are described in Section 6.4.2, which follows. Many other sequential pattern mining applications may not be covered by this model. For example, when analyzing Web clickstream sequences, gaps between clicks become important if one wants to predict what the next click might be. In DNA sequence analysis, *approximate* patterns become useful since DNA sequences may contain (symbol) insertions, deletions, and mutations. Such diverse requirements can be viewed as *constraint relaxation* or *enforcement*. In Section 6.4.3, we discuss how to extend the basic sequential mining model to *constrained* sequential pattern mining in order to handle these cases.

6.4.2 Scalable Methods for Mining Sequential Patterns

Sequential pattern mining is computationally challenging since such mining may generate and/or test a combinatorially explosive number of intermediate subsequences.

"*How can we develop efficient and scalable methods for sequential pattern mining?*" We may categorize the sequential pattern mining methods into two categories: (1) efficient methods for mining the *full set* of sequential patterns, and (2) efficient methods for mining only the *set of closed* sequential patterns, where a sequential pattern s is **closed** if there exists no sequential pattern s' where s' is a proper supersequence of s, and s' has the same (frequency) support as s.[3] Since all of the subsequences of a frequent sequence are also frequent, mining the set of closed sequential patterns may avoid the generation of unnecessary subsequences and thus lead to more compact results as well as more efficient methods than mining the full set. We will first examine methods for mining the full set and then study how they can be extended for mining the closed set. In addition, we discuss modifications for mining multilevel, multidimensional sequential patterns (that is, with multiple levels of granularity).

The major approaches for mining the full set of sequential patterns are similar to those introduced for frequent itemset mining in Chapter 5. Here, we discuss three such approaches for sequential pattern mining, represented by the algorithms GSP, SPADE, and PrefixSpan, respectively. GSP adopts a *candidate generate-and-test* approach using *horizonal data format* (where

[3]Closed frequent itemsets were introduced in Chapter 5. Here, the definition is applied to sequential patterns.

the data are represented as $\langle sequence\_ID : sequence\_of\_itemsets \rangle$, as usual, where each itemset is an event). SPADE adopts a candidate generate-and-test approach using *vertical data format* (where the data are represented as $\langle itemset : (sequence\_ID, event\_ID) \rangle$). The vertical data format can be obtained by transforming from a horizontally formatted sequence database in just one scan. PrefixSpan is a *pattern growth* method, which does not require candidate generation.

All three approaches either directly or indirectly explore the **Apriori property**, stated as follows: *every non-empty subsequence of a sequential pattern is a sequential pattern.* (Recall that for a pattern to be called sequential, it must be frequent. That is, it must satisfy minimum support.) The Apriori property is antimonotonic (or downward-closed) in that, if a sequence cannot pass a test (e.g., regarding minimum support), all of its supersequences will also fail the test. Use of this property to prune the search space can help make the discovery of sequential patterns more efficient.

GSP: A Sequential Pattern Mining Algorithm Based on Candidate Generate-and-Test

GSP (Generalize Sequential Patterns) is a sequential pattern mining method that was developed by Srikant and Agrawal in 1996. It is an extension of their seminal algorithm for frequent itemset mining, known as Apriori (Section 5.2). GSP makes use of the downward-closure property of sequential patterns and adopts a multiple-pass, candidate generate-and-test approach. The algorithm is outlined as follows. In the first scan of the database, it finds all of the frequent items, that is, those with minimum support. Each such item yields a length-1 frequent sequence consisting of that item. Each subsequent pass starts with a *seed set* of sequential patterns—the set of sequential patterns found in the previous pass. This seed set is used to generate new potentially frequent patterns, called *candidate sequences*. Each candidate sequence contains one more item than the seed sequential pattern from which it was generated. Recall that the number of instances of items in a sequence is the *length* of the sequence. So, all of the candidate sequences in a given pass will have the same length. We refer to a sequence with length k as a k-sequence. Let C_k denote the set of candidate k-sequences. A pass over the database finds the support for each candidate k-sequence. The candidates in C_k with at least *min_sup* form L_k, the set of all *frequent* k-sequences. This set then becomes the seed set for the next pass, $k+1$. The algorithm terminates when no new sequential pattern is found in a pass, or no candidate sequence can be generated.

The method is illustrated in the following example.

Example 6.4.1 **GSP: candidate generate-and-test (using horizontal data format).** Suppose we are given the same sequence database, S, of Table 6.4 from Example 6.4.1, with $min\_sup = 2$. Note that the data are represented in horizontal data format. In the first scan ($k = 1$), GSP collects the support for each item. The set of candidate 1-sequences is thus (shown here in

the form of "*sequence* : *support*"): $\langle a\rangle : 4, \langle b\rangle : 4, \langle c\rangle : 3, \langle d\rangle : 3, \langle e\rangle : 3, \langle f\rangle : 3, \langle g\rangle : 1$.

The sequence $\langle g\rangle$ has a support of only 1, and is the only sequence that does not satisfy minimum support. By filtering it out, we obtain the first seed set, $L_1 = \{\langle a\rangle, \langle b\rangle, \langle c\rangle, \langle d\rangle, \langle e\rangle, \langle f\rangle\}$. Each member in the set represents a length-1 sequential pattern. Each subsequent pass starts with the seed set found in the previous pass and uses it to generate new candidate sequences, which are potentially frequent.

Using L_1 as the seed set, this set of 6 length-1 sequential patterns generates a set of $6 \times 6 + \frac{6\times 5}{2} = 51$ candidate sequences of length 2, $C_2 = \{\langle aa\rangle, \langle ab\rangle, \ldots, \langle af\rangle, \langle ba\rangle, \langle bb\rangle, \ldots, \langle ff\rangle, \langle (ab)\rangle, \langle (ac)\rangle, \ldots, \langle (ef)\rangle\}$.

In general, the set of candidates is generated by a self-join of the sequential patterns found in the previous pass (see Section 5.2.1 for details). GSP applies the Apriori property to prune the set of candidates as follows. In the k-th pass, a sequence is a candidate only if each of its length-$(k-1)$ subsequences is a sequential pattern found at the $(k-1)$-th pass. A new scan of the database collects the support for each candidate sequence and finds a new set of sequential patterns, L_k. This set becomes the seed for the next pass. The algorithm terminates when no sequential pattern is found in a pass, or when there is no candidate sequence generated. Clearly, the number of scans is at least the maximum length of sequential patterns. GSP needs one more scan if the sequential patterns obtained in the last scan still generate new candidates.

Although GSP benefits from the Apriori pruning, it still generates a large number of candidates. In this example, 6 length-1 sequential patterns generate 51 length-2 candidates; 22 length-2 sequential patterns generate 64 length-3 candidates; and so on. Some candidates generated by GSP may not appear in the database at all. In this example, 13 out of 64 length-3 candidates do not appear in the database, resulting in wasted search effort.

The example shows that although an Apriori-like sequential pattern mining method, such as GSP, reduces search space, it typically needs to scan the database multiple times. It will likely generate a huge set of candidate sequences, especially when mining long sequences. There is a need for more efficient mining method.

SPADE: An Apriori-Based Vertical Data Format Sequential Pattern Mining Algorithm

The Apriori-like sequential pattern mining approach (based on candidate generate-and-test) can also be explored by mapping a sequence database into vertical data format. In **vertical data format**, the database becomes a set of tuples of the form $\langle itemset : (sequence\_ID, event\_ID)\rangle$. That is, for a given itemset, we record the sequence identifier and corresponding event identifier for which the itemset occurs. The **event identifier** serves as a timestamp within a sequence. The *event_ID* of the ith itemset (or event) in a sequence is i. Note than an itemset can occur in more than one sequence. The set of $(sequence\_ID, event\_ID)$

pairs for a given itemset forms the **ID_list** of the itemset. The mapping from horizontal to vertical format requires one scan of the database. A major advantage of using this format is that we can determine the support of any k-sequence by simply joining the ID_lists of any two of its $(k-1)$-length subsequences. The length of the resulting ID_list (i.e., unique $sequence\_ID$ values) is equal to the support of the k-sequence, which tells us whether or not the sequence is frequent.

SPADE (**S**equential **PA**ttern **D**iscovery using **E**quivalent classes) is an Apriori-based sequential pattern mining algorithm that uses vertical data format. As with GSP, SPADE requires one scan to find the frequent 1-sequences. To find candidate 2-sequences, we join all pairs of single items if they are frequent (therein, it applies the Apriori property), share the same sequence identifier, and their event identifiers follow a sequential ordering. That is, the first item in the pair must occur as an event before the second item, where both occur in the same sequence. Similarly, we can grow the length of itemsets from length 2 to length three, and so on. The procedure stops when no frequent sequences can be found or no such sequences can be formed by such joins. The following example helps illustrate the process.

Example 6.4.2 **SPADE: candidate generate-and-test using vertical data format.** Let $min\_sup = 2$. Our running example sequence database, S, of Table 6.4 is in horizonal data format. SPADE first scans S and transforms it into the vertical format, as shown in Figure 6.8(a). Each itemset (or event) is associated with its ID_list, which is the set of SID ($sequence\_ID$) and EID ($event\_ID$) pairs that contain the itemset. The ID_list for individual items, a, b, and so on, is shown in Figure 6.8(b). For example, the ID_list for item b consists of the following (SID, EID) pairs: $\{(1,2),(2,3),(3,2),(3,5),(4,5)\}$, where the entry (1,2) means that b occurs in sequence 1, event 2, etc. Items a and b are frequent. They can be joined to form the length-2 sequence, $\langle a, b\rangle$. We find the support of this sequence as follows. We join the ID_lists of a and b by joining on the same $sequence\_ID$ wherever, according to the $event_ID$s, a occurs before b. That is, the join must preserve the temporal order of the events involved. The result of such a join for a and b is shown in the ID_list for ab of Figure 6.8(c). For example, the ID_list for 2-sequence ab is a set of triples, $(SID, EID(a), EID(b))$, namely $\{(1,1,2),(2,1,3),(3,2,5),(4,3,5)\}$. The entry (2,1,3), for example, shows that both a and b occur in sequence 2, and that a (event 1 of the sequence) occurs before b (event 3), as required. Furthermore, the frequent 2-sequences can be joined (while considering the pruning heuristic that the (k-1)-subsequences of a candidate k-sequence must be frequent) to form 3-sequences, as in Figure 6.8(d), and so on. The process terminates when no frequent sequences can be found or no candidate sequences can be formed.

The use of vertical data format, with the creation of ID_lists, reduces scans of the sequence database. The ID_lists carry the information necessary to find the support of candidates. As the length of a frequent sequence increases, the size of its ID_list decreases, resulting in fast joins. However, the basic search methodology of SPADE and GSP is breadth-first search (e.g., exploring 1-sequences, then 2-sequences, and so on) and Apriori pruning. Despite the pruning, both

SID	*EID*	*itemset*
1	1	a
1	2	abc
1	3	ac
1	4	d
1	5	cf
2	1	ad
2	2	c
2	3	bc
2	4	ae
3	1	ef
3	2	ab
3	3	df
3	4	c
3	5	b
4	1	e
4	2	g
4	3	af
4	4	c
4	5	b
4	6	c

(a) vertical format database

a		b		...
SID	*EID*	*SID*	*EID*	...
1	1	1	2	
1	2	2	3	
1	3	3	2	
2	1	3	5	
2	4	4	5	
3	2			
4	3			

(b) ID_lists for some 1-sequences

ab			ba			...
SID	*EID*(*a*)	*EID*(*b*)	*SID*	*EID*(*b*)	*EID*(*a*)	...
1	1	2	1	2	3	
2	1	3	2	3	4	
3	2	5				
4	3	5				

(c) ID_lists for some 2-sequences

aba				...
SID	*EID*(*a*)	*EID*(*b*)	*EID*(*a*)	...
1	1	2	3	
2	1	3	4	

(d) ID_lists for some 3-sequences

Figure 6.8: The mining process: (a) vertical format database; (b) to (d) show fragments of the ID_lists for 1-sequences, 2-sequences, and 3-sequences, respectively. [TO EDITOR Please kindly do some table spacing and center adjustments. Thanks.]

algorithms have to generate large sets of candidates in breadth-first manner in order to grow longer sequences. Thus, most of the difficulties suffered in the GSP algorithm will reoccur in SPADE as well.

PrefixSpan: Prefix-Projected Sequential Pattern Growth

Pattern growth is a method of frequent-pattern mining that does not require candidate generation. The technique originated in the FP-growth algorithm for transaction databases, presented in Section 5.6. The general idea of this approach is as follows: it finds the frequent single items, then compresses this information into a *frequent-pattern tree*, or *FP-tree*. The FP-tree is used to generation a set of projected databases, each associated with one frequent item. Each of these databases is mined separately and recursively, avoiding candidate generation. Interestingly, the pattern-growth approach can be extended to min-

ing sequential patterns, which leads to a new algorithm, PrefixSpan, illustrated below.

Without loss of generality, all the items within an event can be listed alphabetically. For example, instead of listing the items in an event as, say, (bac), we can list them as (abc). Given a sequence $\alpha = \langle e_1 e_2 \cdots e_n \rangle$ (where each e_i corresponds to a frequent event in a sequence database, S), a sequence $\beta = \langle e'_1 e'_2 \cdots e'_m \rangle$ $(m \leq n)$ is called a **prefix** of α if and only if (1) $e'_i = e_i$ for $(i \leq m-1)$; (2) $e'_m \subseteq e_m$; and (3) all the frequent items in $(e_m - e'_m)$ are alphabetically after those in e'_m. Sequence $\gamma = \langle e''_m e_{m+1} \cdots e_n \rangle$ is called the **suffix** of α with respect to prefix β, denoted as $\gamma = \alpha/\beta$, where $e''_m = (e_m - e'_m)$.[4] We also denote $\alpha = \beta \cdot \gamma$. Note if β is not a subsequence of α, the suffix of α with respect to β is empty.

We illustrate these concepts with the following example.

Example 6.4.3 **Prefix and suffix.** Let sequence $s = \langle a(abc)(ac)d(cf) \rangle$, which corresponds to sequence 1 of our running example sequence database. $\langle a \rangle$, $\langle aa \rangle$, $\langle a(ab) \rangle$ and $\langle a(abc) \rangle$ are four prefixes of s. $\langle (abc)(ac)d(cf) \rangle$ is the suffix of s with respect to the prefix $\langle a \rangle$; $\langle (\_bc)(ac)d(cf) \rangle$ is its suffix with respect to the prefix $\langle aa \rangle$; and $\langle (\_c)(ac)d(cf) \rangle$ is its suffix with respect to the prefix $\langle a(ab) \rangle$.

Based on the concepts of prefix and suffix, the problem of mining sequential patterns can be decomposed into a set of subproblems as shown below.

1. Let $\{\langle x_1 \rangle, \langle x_2 \rangle, \ldots, \langle x_n \rangle\}$ be the complete set of length-1 sequential patterns in a sequence database, S. The complete set of sequential patterns in S can be partitioned into n disjoint subsets. The i^{th} subset $(1 \leq i \leq n)$ is the set of sequential patterns with prefix $\langle x_i \rangle$.

2. Let α be a length-l sequential pattern and $\{\beta_1, \beta_2, \ldots, \beta_m\}$ be the set of all length-$(l+1)$ sequential patterns with prefix α. The complete set of sequential patterns with prefix α, except for α itself, can be partitioned into m disjoint subsets. The j^{th} subset $(1 \leq j \leq m)$ is the set of sequential patterns prefixed with β_j.

Based on this observation, the problem can be partitioned recursively. That is, each subset of sequential patterns can be further partitioned when necessary. This forms a *divide-and-conquer* framework. To mine the subsets of sequential patterns, we construct corresponding *projected databases* and mine each one recursively.

Let's use our running example to examine how to use the prefix-based projection approach for mining sequential patterns.

Example 6.4.4 **PrefixSpan: A pattern-growth approach.** Using the same sequence database, S, of Table 6.4 with $min\_sup = 2$, sequential patterns in S can be mined by a prefix-projection method in the following steps.

[4] If e''_m is not empty, the suffix is also denoted as $\langle (\_ \text{ items in } e''_m) e_{m+1} \cdots e_n \rangle$.

1. *Find length-1 sequential patterns.* Scan S once to find all of the frequent items in sequences. Each of these frequent items is a length-1 sequential pattern. They are $\langle a\rangle : 4$, $\langle b\rangle : 4$, $\langle c\rangle : 4$, $\langle d\rangle : 3$, $\langle e\rangle : 3$, and $\langle f\rangle : 3$, where the notation "$\langle pattern\rangle : count$" represents the pattern and its associated support count.

2. *Partition the search space.* The complete set of sequential patterns can be partitioned into the following six subsets according to the six prefixes: (1) the ones with prefix $\langle a\rangle$, (2) the ones with prefix $\langle b\rangle$, ..., and (6) the ones with prefix $\langle f\rangle$.

prefix	projected database	sequential patterns
$\langle a\rangle$	$\langle(abc)(ac)d(cf)\rangle$, $\langle(\_d)c(bc)(ae)\rangle$, $\langle(\_b)(df)cb\rangle$, $\langle(\_f)cbc\rangle$	$\langle a\rangle$, $\langle aa\rangle$, $\langle ab\rangle$, $\langle a(bc)\rangle$, $\langle a(bc)a\rangle$, $\langle aba\rangle$, $\langle abc\rangle$, $\langle(ab)\rangle$, $\langle(ab)c\rangle$, $\langle(ab)d\rangle$, $\langle(ab)f\rangle$, $\langle(ab)dc\rangle$, $\langle ac\rangle$, $\langle aca\rangle$, $\langle acb\rangle$, $\langle acc\rangle$, $\langle ad\rangle$, $\langle adc\rangle$, $\langle af\rangle$
$\langle b\rangle$	$\langle(\_c)(ac)d(cf)\rangle$, $\langle(\_c)(ae)\rangle$, $\langle(df)cb\rangle$, $\langle c\rangle$	$\langle b\rangle$, $\langle ba\rangle$, $\langle bc\rangle$, $\langle(bc)\rangle$, $\langle(bc)a\rangle$, $\langle bd\rangle$, $\langle bdc\rangle$, $\langle bf\rangle$
$\langle c\rangle$	$\langle(ac)d(cf)\rangle$, $\langle(bc)(ae)\rangle$, $\langle b\rangle$, $\langle bc\rangle$	$\langle c\rangle$, $\langle ca\rangle$, $\langle cb\rangle$, $\langle cc\rangle$
$\langle d\rangle$	$\langle(cf)\rangle$, $\langle c(bc)(ae)\rangle$, $\langle(\_f)cb\rangle$	$\langle d\rangle$, $\langle db\rangle$, $\langle dc\rangle$, $\langle dcb\rangle$
$\langle e\rangle$	$\langle(\_f)(ab)(df)cb\rangle$, $\langle(af)cbc\rangle$	$\langle e\rangle$, $\langle ea\rangle$, $\langle eab\rangle$, $\langle eac\rangle$, $\langle eacb\rangle$, $\langle eb\rangle$, $\langle ebc\rangle$, $\langle ec\rangle$, $\langle ecb\rangle$, $\langle ef\rangle$, $\langle efb\rangle$, $\langle efc\rangle$, $\langle efcb\rangle$.
$\langle f\rangle$	$\langle(ab)(df)cb\rangle$, $\langle cbc\rangle$	$\langle f\rangle$, $\langle fb\rangle$, $\langle fbc\rangle$, $\langle fc\rangle$, $\langle fcb\rangle$

Table 6.5: Projected databases and sequential patterns

3. *Find subsets of sequential patterns.* The subsets of sequential patterns mentioned in step 2 can be mined by constructing corresponding *projected databases* and mining each recursively. The projected databases, as well as the sequential patterns found in them, are listed in Table 6.5, while the mining process is explained as follows.

   (a) *Find sequential patterns with prefix* $\langle a\rangle$. Only the sequences containing $\langle a\rangle$ should be collected. Moreover, in a sequence containing $\langle a\rangle$, only the subsequence prefixed with the first occurrence of $\langle a\rangle$ should be considered. For example, in sequence $\langle(ef)(ab)(df)cb\rangle$, only the subsequence $\langle(\_b)(df)cb\rangle$ should be considered for mining sequential patterns prefixed with $\langle a\rangle$. Notice that $(\_b)$ means that the last event in the prefix, which is a, together with b, form one event.

   The sequences in S containing $\langle a\rangle$ are projected with respect to $\langle a\rangle$

to form the $\langle a \rangle$-*projected database*, which consists of four suffix sequences: $\langle (abc)(ac)d(cf) \rangle$, $\langle (\_d)c(bc)(ae) \rangle$, $\langle (\_b)(df)cb \rangle$ and $\langle (\_f)cbc \rangle$.

By scanning the $\langle a \rangle$-projected database once, its locally frequent items are $a : 2$, $b : 4$, $\_b : 2$, $c : 4$, $d : 2$, and $f : 2$. Thus all the length-2 sequential patterns prefixed with $\langle a \rangle$ are found, and they are: $\langle aa \rangle : 2$, $\langle ab \rangle : 4$, $\langle (ab) \rangle : 2$, $\langle ac \rangle : 4$, $\langle ad \rangle : 2$, and $\langle af \rangle : 2$.

Recursively, all sequential patterns with prefix $\langle a \rangle$ can be partitioned into 6 subsets: (1) those prefixed with $\langle aa \rangle$, (2) those with $\langle ab \rangle$, ..., and finally, (6) those with $\langle af \rangle$. These subsets can be mined by constructing respective projected databases and mining each recursively as follows.

i. The $\langle aa \rangle$-projected database consists of two non-empty (suffix) subsequences prefixed with $\langle aa \rangle$: $\{\langle (\_bc)(ac)d(cf) \rangle, \{\langle (\_e) \rangle\}$. Since there is no hope of generating any frequent subsequence from this projected database, the processing of the $\langle aa \rangle$-projected database terminates.

ii. The $\langle ab \rangle$-projected database consists of three suffix sequences: $\langle (\_c)(ac)d(cf) \rangle$, $\langle (\_c)a \rangle$, and $\langle c \rangle$. Recursively mining the $\langle ab \rangle$-projected database returns four sequential patterns: $\langle (\_c) \rangle$, $\langle (\_c)a \rangle$, $\langle a \rangle$, and $\langle c \rangle$ (i.e., $\langle a(bc) \rangle$, $\langle a(bc)a \rangle$, $\langle aba \rangle$, and $\langle abc \rangle$.) They form the complete set of sequential patterns prefixed with $\langle ab \rangle$.

iii. The $\langle (ab) \rangle$-projected database contains only two sequences: $\langle (\_c)(ac)d(cf) \rangle$ and $\langle (df)cb \rangle$, which leads to the finding of the following sequential patterns prefixed with $\langle (ab) \rangle$: $\langle c \rangle$, $\langle d \rangle$, $\langle f \rangle$, and $\langle dc \rangle$.

iv. The $\langle ac \rangle$-, $\langle ad \rangle$- and $\langle af \rangle$- projected databases can be constructed and recursively mined in a similar manner. The sequential patterns found are shown in Table 6.5.

(b) *Find sequential patterns with prefix* $\langle b \rangle$, $\langle c \rangle$, $\langle d \rangle$, $\langle e \rangle$ *and* $\langle f \rangle$, *respectively.* This can be done by constructing the $\langle b \rangle$-, $\langle c \rangle$- $\langle d \rangle$-, $\langle e \rangle$- and $\langle f \rangle$-projected databases and mining them respectively. The projected databases as well as the sequential patterns found are also shown in Table 6.5.

4. *The set of sequential patterns is the collection of patterns found in the above recursive mining process.*

The method described above generates no candidate sequences in the mining process. However, it may generate many projected databases, one for each frequent prefix-subsequence. Forming a large number of projected databases recursively may become the major cost of the method, if such databases have to be generated physically. An important optimization technique is **pseudo-projection**, which registers the index (or identifier) of the corresponding sequence and the starting position of the projected suffix in the sequence instead of performing physical projection. That is, a physical projection of a sequence

is replaced by registering a sequence identifier and the projected position index point. Pseudo-projection reduces the cost of projection substantially when such projection can be done in main memory. However, it may not be efficient if the pseudo-projection is used for disk-based accessing since random access of disk space is costly. The suggested approach is that if the original sequence database or the projected databases are too big to fit in memory, the physical projection should be applied, however, the execution should be swapped to pseudo-projection once the projected databases can fit in memory. This methodology is adopted in the PrefixSpan implementation.

A performance comparison of GSP, SPADE, and PrefixSpan shows that PrefixSpan has the best overall performance. SPADE, though weaker than PrefixSpan in most cases, outperforms GSP. Generating huge candidate sets may consume a tremendous amount of memory, thereby causing candidate generate-and-test algorithms to become rather slow. The comparison also found that when there is a large number of frequent subsequences, all three algorithms run slowly. This problem can be partially solved by closed sequential pattern mining.

Mining Closed Sequential Patterns

Since mining the complete set of frequent subsequences can generate a huge number of sequential patterns, an interesting alternative is to mine frequent *closed subsequences* only, that is, those containing no supersequence with the same support. Mining closed sequential patterns can produce a significantly less number of sequences than the full set of sequential patterns. Note that the full set of frequent subsequences, together with their supports, can easily be derived from the closed subsequences. Thus, closed subsequences have the same expressive power as the corresponding full set of subsequences. Because of their compactness, they may also be quicker to find.

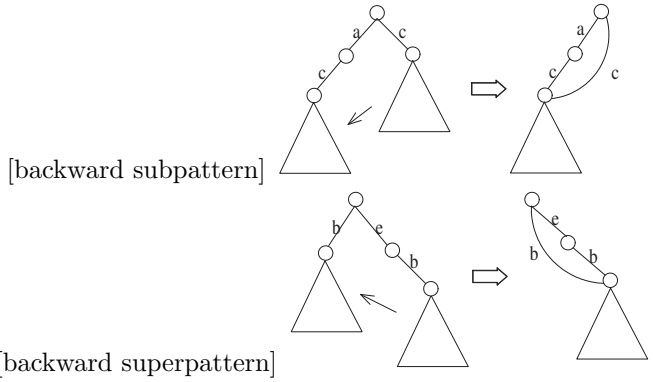

Figure 6.9: The pruning of a backward subpattern or a backward superpattern.

CloSpan is an efficient closed sequential pattern mining method. Similar to mining closed frequent patterns, it can skip mining redundant closed sequential

pattern if it finds the continuous mining will not generate any new results. For example, CloSpan will prune the search for *backward subpatterns* and *backward superpatterns*, if it will lead to redundant mining, as indicated in Figure 6.9. More concretely, it will stop growing a prefix-based projected databases $SDB|_\beta$ if it is of the same size as that of the prefix-based projected database $SDB|_\alpha$ and α and β have substring/superstring relationships.

This is based on the based on a property of sequence databases, called **equivalence of projected databases**, stated as follows: *Two projected sequence databases,* $SDB|_\alpha = S|_\beta$,[5] $\alpha \sqsubseteq \beta$ *(i.e.,* α *is a subsequence of* β*), are equivalent if and only if the total number of items in* $SDB|_\alpha$ *is equal to the total number of items in* $SDB|_\beta$.

Let's examine one such example.

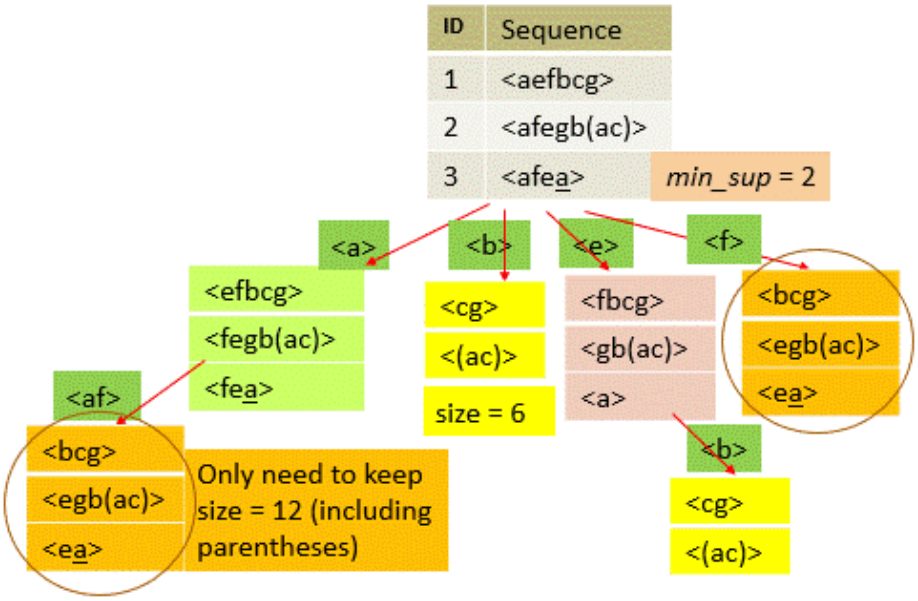

Figure 6.10: The pruning of a backward subpattern or a backward superpattern.

Example 6.4.5 **CloSpan: Pruning redundant projected database.** Given a small sequence database, SDB, shown in Figure 6.10, with $min\_sup = 2$. The prefix project sequence database of the prefix $\langle af \rangle$ is: $(\langle acg \rangle, \langle egb(ac) \rangle, \langle ea \rangle)$ with 12 symbols (including parentheses), and the projected sequence database of the prefix $\langle f \rangle$ is of the same size. Clearly, the two projected databases should be identical and there is no need to mine the latter, the $\langle f \rangle$-projected sequence database. This is understandable since for any sequence s, if its projections on $\langle af \rangle$ and $\langle f \rangle$ respectively are not identical, the latter must contain more

[5] In $SDB|_\alpha$, a sequence database SDB is projected with respect to sequence (e.g., prefix) α. The notation $SDB|_\beta$ can be similarly defined.

symbols than the former (e.g., it may contain only $\langle f \rangle$ but not $\langle a \dots f \rangle$ or has $\langle f \rangle$ in front of $\langle a \dots f \rangle$). But now, the two sizes are equal, then their projected databases must be identical. Such backward subpattern pruning and backward superpattern pruning can reduce the search space substantially.

Empirical results show that CloSpan often derives a much smaller set of sequential patterns in a shorter time than PrefixSpan, which mines the complete set of sequential patterns.

Mining Multidimensional, Multilevel Sequential Patterns

Sequence identifiers (representing individual customers, for example) and sequence items (such as products bought) are often associated with additional pieces of information. Sequential pattern mining may take advantage of such additional information to discover interesting patterns in multidimensional, multilevel information space. Take customer shopping transactions, for instance. In a sequence database for such data, the additional information associated with sequence IDs could include customer residential area, group, and profession. Information associated with items could include item category, brand, model type, model number, place manufactured, and manufacture date. Mining *multidimensional, multilevel* sequential patterns is the discovery of interesting patterns in such a broad dimensional space, at different levels of detail.

Example 6.4.6 **Multidimensional, multilevel sequential patters.** The discovery that "*Retired customers who purchase a smart home thermostat are likely to purchase a video doorbell within a month*" and that "*Young adults who purchase a laptop are likely to buy laser printer within 90 days*" are examples of multidimensional, multilevel sequential patterns. By grouping customers into *"retired customers"* and *"young adults"* according to the values in the age dimension, and by generalizing items to, say, *"smart thermostat"* rather than a specific model, the patterns mined here are associated with certain dimensions and are at a higher level of abstraction.

"Can a typical sequential pattern algorithm such as PrefixSpan be extended to efficiently mine multidimensional, multilevel sequential patterns?" One suggested modification is to associate the multidimensional, multilevel information with the *sequence_ID* and *item_ID*, respectively, which the mining method can take into consideration when finding frequent subsequences. For example, $(Chicago, middle\_aged, business)$ can be associated with *sequence_ID_1002* (for a given customer) whereas $(laserprinter, HP, LaserJetPro, G3Q47A, USA, 2020)$ can be associated with *item_ID_543005* in the sequence. A sequential pattern mining algorithm will use such information in the mining process to find sequential patterns associated with multidimensional, multilevel information.

6.4.3 Constraint-Based Mining of Sequential Patterns

As shown in our study of frequent-pattern mining, mining that is performed without user- or expert-specified constraints may generate numerous patterns

that are of no interest. Such unfocused mining can reduce both the efficiency and usability of frequent-pattern mining. Thus, we promote **constraint-based mining**, which incorporates user-specified constraints to reduce the search space and derive only patterns that are of interest to the user.

Constraints can be expressed in many forms. They may specify desired relationships between attributes, attribute values, or aggregates within the resulting patterns mined. Regular expressions can also be used as constraints in the form of "pattern templates", which specify the desired form of the patterns to be mined. The general concepts introduced for constraint-based frequent pattern mining apply to constraint-based sequential pattern mining as well. The key idea to note is that these kinds of constraints can be used *during* the mining process to confine the search space, thereby improving (1) the efficiency of the mining, and (2) the interestingness of the resulting patterns found. This idea is also referred to as *"pushing the constraints deep into the mining process."*

We now examine some typical examples of constraints for sequential pattern mining.

First, constraints can be related to the **duration**, T, of a sequence. The duration can be user-specified, related to a particular time period, such as within the last six months. Sequential pattern mining can then be confined to the data within the specified duration, T. Constraints related to a specific duration, can be considered as *succinct* constraints. A constraint is **succinct** if we can enumerate all and only those sequences that are guaranteed to satisfy the constraint, even before support counting begins. In this case, we can push the data selection process deeply into the mining process, select sequences in the desired period before mining begins to reduce the search space.

Second, a user may confine the maximal or minimal length of the sequential patterns to be mined. The maximal or minimal length of sequential patterns can be treated as *antimonotonic* or *monotonic* constraints, respectively. For example, the constraint $L \leq 10$ is *antimonotonic* since, if a sequential pattern violates this constraint, further mining following it will always violate the constraint. Similarly, data anti-monoticity and its search space pruning rules can be established correspondingly for sequential pattern mining as well.

Third, in sequential pattern mining, a constraint can be related to an **event folding window**, w. A set of events occurring within a specified period of time can be viewed as occurring together. If w is set to 0 (i.e., no event sequence folding), sequential patterns are found where each event occurs at a distinct time instant, such as "*a customer bought a laptop, then a digital camera, and then a laser printer*" will be considered as a length-3 sequence, even if all these happen within the same day. However, if w is set to be weekly based, then these transactions are considered as occurring within the same period, and such sequences are "folded" in the analysis. On the extreme, if w is set to be as long as the whole duration, T, sequential pattern mining is degenerated into sequence-insensitive frequent pattern mining.

Fourth, a desired time **gap** between events in the discovered patterns may be specified as a constraint. For example, $min\_gap \leq gap \leq max\_gap$ is to find patterns that are separated by at least min_gap but at most max_gap. A pattern

like "*If a person rents movie A, it is likely she will rent movie B not within 6 days but within 30 days*" implies $6 < gap \leq 30$ (days). It is straightforward to push gap constraints into the sequential pattern mining process. With minor modifications to the mining process, it can handle constraints with approximate gaps as well.

Finally, a user can specify constraints on the kinds of sequential patterns by providing "pattern templates" in the form of regular expressions. Here we discuss mining *serial episodes* and *parallel episodes* using *regular expressions*. A **serial episode** is a set of events that occurs in a total order, whereas a **parallel episode** is a set of events whose occurrence ordering is trivial. Consider the following example.

Example 6.4.1 **Specifying serial episodes and parallel episodes with regular expressions.** Let the notation (E, t) represent *event type* E at *time* t. Consider the data $(A, 1)$, $(C, 2)$, and $(B, 5)$ with an event folding window width of $w = 2$, where the serial episode $A \rightarrow B$ and the parallel episode $A \,\&\, C$ both occur in the data. The user can specify constraints in the form of a regular expression, such as $\{A|B\}C * \{D|E\}$, which indicates that the user would like to find patterns where event A and B first occur (but they are parallel in that their relative ordering is unimportant), followed by one or a set of events C, followed by the events D and E (where D can occur either before or after E). Other events can occur in between those specified in the regular expression.

A regular expression constraint may be neither antimonotonic nor monotonic. In such cases, we cannot use it to prune the search space in the same ways as described above. However, by modifying the PrefixSpan-based pattern-growth approach, such constraints can be handled in an elegant manner. Let's examine one such example.

Example 6.4.2 **Constraint-based sequential pattern mining with a regular expression constraint.** Suppose that our task is to mine sequential patterns, again using the sequence database, S, of Table 6.4. This time, however, we are particularly interested in patterns that match the regular expression constraint, $C = \langle a * \{bb|(bc)d|dd\}\rangle$, with minimum support.

Such a regular expression constraint is neither antimonotonic, nor monotonic, nor succinct. Therefore, it cannot be pushed deep into the mining process. Nonetheless, this constraint can easily be integrated with the pattern-growth mining process as follows. First, only the $\langle a\rangle$-projected database, $S|_{\langle a\rangle}$, needs to be mined since the regular expression constraint C starts with a. Retain only the sequences in $S|_{\langle a\rangle}$ that contain items within the set $\{b, c, d\}$. Second, the remaining mining can proceed from the suffix. This is essentially the *Suffix-Span* algorithm, which is symmetric to PrefixSpan in that it grows suffixes from the end of the sequence forward. The growth should match the suffix as the constraint, $\langle\{bb|(bc)d|dd\}\rangle$. For the projected databases that match these suffixes, we can grow sequential patterns either in prefix- or suffix- expansion manner to find all of the remaining sequential patterns.

Thus, we have seen several ways in which constraints can be used to improve the efficiency and usability of sequential pattern mining.

6.5 Mining Subgraph Patterns

Graphs become increasingly important in modeling complicated structures, such as circuits, images, workflows, XML documents, webpages, chemical compounds, protein structures, biological networks, social networks, information networks, knowledge graphs, and the Web. Many graph search algorithms have been developed in chemical informatics, computer vision, video indexing, Web search, 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* or *subgraphs* 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 sub-pathways in metabolic networks, and on the use of frequent graph patterns for graph indexing and similarity search in graph databases. Although graph mining may include mining frequent subgraph patterns, graph classification, clustering, and other analysis tasks, in this section we focus on mining frequent subgraphs. We looks at various methods, their extensions, and applications.

6.5.1 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 a graph database (i.e., a collection of graphs) 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 6.5.1 **Frequent subgraph.** Figure 6.11 shows a sample set of chemical structures. Figure 6.12 depicts two of the frequent subgraphs in this data set, given a minimum support of 66.6%.

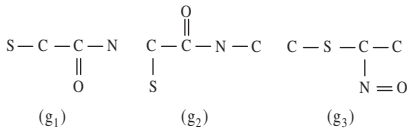

Figure 6.11: A sample graph data set.

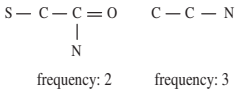

Figure 6.12: Frequent graphs.

"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. This is because the second step involves a subgraph isomorphism test whose computational complexity is excessively high (that is, 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-grown approach.

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 6.13. 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

Algorithm: AprioriGraph(D, minsup, S_k)

Input: a graph data set D, and .
Output: The frequent substructure set S_k.

```
1: S_{k+1} ← ∅;
2: for each frequent g_i ∈ S_k do
3:   for each frequent g_j ∈ 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 ∉ S_{k+1} then
6:         insert g to S_{k+1};
7: if s_{k+1} ≠ ∅ then
8:   call AprioriGraph(D, minsup, S_{k+1});
9: return;
```

Figure 6.13: AprioriGraph.

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, since there are many ways to join two substructures, as shown below.

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. Since edge is a bigger unit than vertex and it enforces more constraints than single vertex, the *edge-based candidate generation method* FSG leads to improved efficiency over the *vertex-based candidate generation* method, AGM. We examine the FSG method here.

The FSG algorithm adopts an *edge-based candidate generation* strategy that increases the substructure size by one edge in each call of AprioriGraph. Two size-k patterns are merged if and only if they share the same subgraph having $k-1$ edges, which is called the **core**. Here, *graph size* is taken to be the number of edges in the graph. The newly formed candidate includes the core and the additional two edges from the size-k patterns. Figure 6.14 shows potential

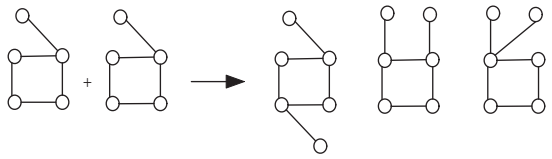

Figure 6.14: FSG: Two substructure patterns and their potential candidates.

candidates formed by two structure patterns. Each candidate has one more edge than these two patterns, but this additional edge can be associated with different

vertices. This example illustrates the complexity of joining two structures to form a large pattern candidate.

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

Apriori-based algorithms have considerable overhead when joining two size-k frequent substructures to generate size-$(k+1)$ graph candidates. The overhead occurs when (1) joining two size-k frequent graphs (or other structures like paths) to generate size-$(k+1)$ graph candidates, and (2) checking the frequency of these candidates separately. These two operations constitute the performance bottlenecks of the Apriori-like algorithms. In order to avoid such overhead, non-Apriori-based algorithms have 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 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.

Algorithm: PatternGrowthGraph(g, D, minsup, S)

Input: A frequent graph g, a graph data set D, and the support threshold *minsup*.
Output: The frequent graph set S.

1: **if** $g \in S$ **then return**;
2: **else** insert g to 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: Call PatternGrowthGraph($g \diamond_x e$, D, minsup, S);
6: **return**;

Figure 6.15: PatternGrowthGraph.

Figure 6.15 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 at the 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 generation of duplicate graphs, each frequent graph should be extended as conservatively as possible. This principle leads to the design of several new algorithms. A typical such example is the gSpan algorithm as described below.

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

Let's see how the gSpan algorithm works. To traverse graphs, it adopts depth-first search. Initially, a starting vertex is randomly chosen and the vertices in a graph are marked so that we can tell which vertices have been visited. The visited vertex set is expanded repeatedly until a full depth-first search (DFS) tree is built. One graph may have various DFS trees depending on how the depth-first search is performed, that is, the vertex visiting order. The darkened edges in Figures 6.16(b) to 6.16(d) show three DFS trees for the same graph of Figure 6.16(a). The vertex labels are x, y, and z; the edge labels are a and

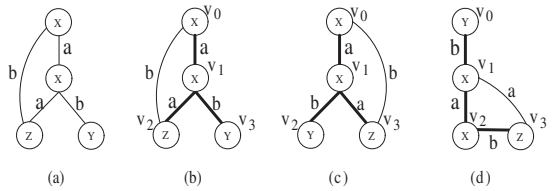

Figure 6.16: DFS subscripting.

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*, and the last visited vertex, v_n, the *right-most vertex*. The straight path from v_0 to v_n is called the *right-most path*.

In Figures 6.16(b) to 6.16(d), three different subscriptings are generated based on the corresponding DFS trees. The right-most path is (v_0, v_1, v_3) in Figures 6.16(b) and 6.16(c), and (v_0, v_1, v_2, v_3) in Figure 6.16(d).

PatternGrowth extends a frequent graph in every possible position, which may generate a large number of duplicate graphs. The gSpan algorithm introduces a more sophisticated extension method. The new method restricts the extension as follows: Given a graph G and a DFS tree T in G, a new edge e can be added between the right-most vertex and other vertices on the right-most path (*backward extension*); or it can introduce a new vertex and connect to vertices on the right-most path (*forward extension*). Since 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 6.5.2 **Backward extension and forward extension.** If we want to extend the graph in Figure 6.16(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.

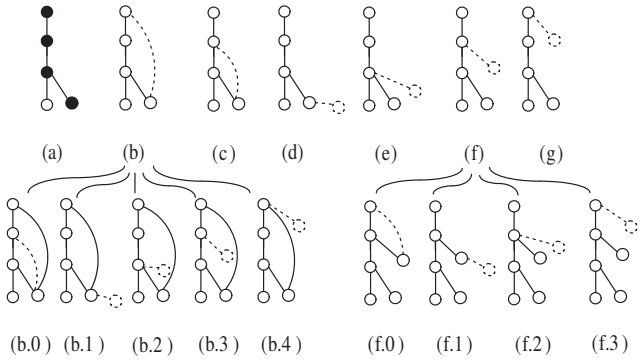

Figure 6.17: Right-Most Extension

Figures 6.17(b) to 6.17(g) show all the potential right-most extensions of Figure 6.17(a). The darkened vertices show the rightmost path. Among these, Figures 6.17(b) to 6.17(d) grow from the rightmost vertex while Figures 6.17(e) to 6.17(g) grow from other vertices on the rightmost path. Figures 6.17(b.0) to 6.17(b.4) are children of Figure 6.17(b), and Figures 6.17(f.0) to 6.17(f.3) are children of Figure 6.17(f). In summary, backward extension only takes place on the rightmost vertex while forward extension introduces a new edge from vertices on the rightmost path.

Since 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.

First, we introduce edge order. Intuitively, DFS tree defines the discovery order of forward edges. For the graph shown in Figure 6.16(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 6.16(b), its backward edge $(2,0)$ should appear after $(1,2)$ since 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 6.16(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 Figures 6.16(b), 6.16(c) and 6.16(d) respectively, are shown in Table 6.6. 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.

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

Table 6.6: DFS code for Figure 6.16(b), 6.16(c), and 6.16(d).

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. Since 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 exactly 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 6.6 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 *DFS Lexicographic Order*. According to this ordering, we have $\gamma_0 < \gamma_1 < \gamma_2$ for the DFS codes listed in Table 6.6.

Based on the DFS lexicographic ordering, the *minimum DFS code* of a given graph G, written as (G), is the minimal one among all the DFS codes. For example, code γ_0 in Table 6.6 is the minimum DFS code of the graph in Figure 6.16(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 $(G) = (G^{'})$*. 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.

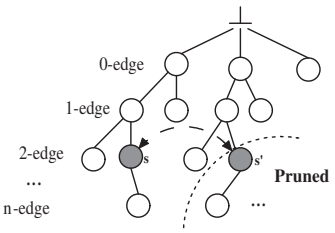

Figure 6.18: Lexicographic search tree.

Figure 6.18 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. Since 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 non-minimum. The search of non-minimum DFS codes does not produce a useful result. *"Is it necessary to perform right-most extension on non-minimum DFS codes?"* The answer is "*no*". If codes s and s' in Figure 6.18 encode the same graph, the search space under s' can be safely pruned.

The details of gSpan are depicted in Figure 6.19. gSpan is called recursively to extend graph patterns so that their frequent descendants are found until their support is lower than minsup or its code is not minimum any more. The difference between gSpan and PatternGrowth is at the right-most extension and

extension termination of non-minimum 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.

Algorithm: gSpan(s, D, minsup, S)

Input: A DFS code s, a graph data set D, and .
Output: The frequent graph set 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: Call gSpan($s \diamond_r e$, D, minsup, S);
9: **return**;

Figure 6.19: gSpan: An pattern-growth algorithm for frequent substructure mining.

The algorithm of Figure 6.19 implements a depth-first search version of gSpan. Actually, breadth-first search works too: for each newly discovered frequent subgraph 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.

6.5.2 Mining Variant and Constrained Substructure Patterns

The frequent subgraph mining discussed in the previous section handles only one special kind of graph: *labeled, undirected, connected simple graphs without any specific constraints*. That is, we assume that the database to be mined contains a set of graphs each consisting of a set of labeled vertices and labeled but undirected edges, with no other constraints. However, many applications or users may need to enforce various kinds of *constraints* on the patterns to be mined or seek *variant substructure patterns*. For example, we may like to mine patterns, each of which 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 organize variants and constraints and help develop efficient mining methods systematically. 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 subgraph as an example. Similar to mining frequent itemsets and mining sequential patterns, mining graph patterns may generate an explosive number of patterns. According to the Apriori property, all the subgraphs of a frequent graph are frequent. Thus 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 1,000,000 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 does not exist a 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 does not exist a 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 6.5.1 **Maximal frequent graph.** The two graphs in Figure 6.12 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 one 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. The key for efficient mining of close frequent subgraphs is to figure out at what condition that the further growth of a frequent subgraph g should be pruned when its e-expanded subgraph g' has the same support as g. Experimental study has shown that CloseGraph often generates far fewer graph patterns and runs more efficiently than gSpan, which mines the full set 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 be extended easily 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 non-simple graphs**. A **non-simple 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 non-simple graphs efficiently too.

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

Fourth, the method can also be extended to **mining disconnected graphs**. There are two cases to be considered: (1) the graphs in the data set may be disconnected, and (2) the graph patterns may be disconnected. For the first case, we can transform the original data set by adding a virtual vertex to connect the disconnected graphs in each graph. We then apply gSpan on the new graph data set. For the second case, we redefine the DFS code. A disconnected graph pattern can be viewed as a set of connected graphs, $r = \{g_0, g_1, \ldots, g_m\}$, where g_i is a connected graph, $0 \leq i \leq m$. Since 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 (since 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.

Mining Substructure Patterns with User-Specified Constraints

Various kinds of constraints or specific requirements 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 to facilitate such mining.

Constraint-based mining of frequent substructures. Constraint-based mining of frequent substructures can be developed systematically, similar to the constraint-based mining of frequent patterns and sequential patterns introduced previously. Take graph mining as an example. As with the constraint-based frequent pattern mining framework, graph constraints can also be classified into a few categories, including *pattern antimonotonic, pattern monotonic, data antimonotonic*, 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 6.5.2 **Constraint-based substructure mining.** Let's examine a few commonly encountered classes of constraints to see how the constraint-pushing technique can be integrated into the pattern-growth mining framework.

1. **Element, set, or subgraph containment constraint**. Suppose a user requires that the mined pattern contain a particular set of subgraphs. This is a **succinct constraint** which can be pushed deeply 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 such given set of subgraphs. A similar strategy can be developed for 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 a **pattern antimonotonic constraint** since 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. C_G is also a **data antimonotonic constraint**: for any data graph g_i, with respect to a candidate subgraph g_c, if there is no component in the remaining g_i containing edges satisfying C_G, g_i should not be further considered for g_c since it will not support g_c's further expansion.

3. **Value-sum constraint**. One example of such a constraint can be that the sum of (positive) weights on the edges Sum_e be within a range from low to $high$. This constraint can be split into two constraints, $Sum_e \geq low$ and $Sum_e \leq high$. The former is a **pattern monotonic constraint**, since once it is satisfied, further "growth" on the graph by adding more edges will always satisfy the constraint. The latter is a **pattern antimonotonic constraint**, because once the condition it is not satisfied, further growth of Sum_e will never satisfy it. Both constraints are **data antimonotonic** in the sense that any data graph that cannot satisfy these constraints during the pattern growth process should be pruned. The constraint pushing strategy can then be worked out easily.

Notice that a graph-mining query may contain multiple constraints. For example, we may want to mine graph patterns satisfying constraints on both the geometry and the range of the 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 can 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—that is, patterns that are similar to each other and have the 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.

6.6 Summary

- The **scope** of frequent pattern mining research reaches far beyond the basic concepts and methods introduced in Chapter 6 for mining frequent itemsets and associations. This chapter presented a road map of the field, where topics are organized with respect to the kinds of patterns and rules that can be mined, mining methods, and applications.

- In addition to mining for basic frequent itemsets and associations, **advanced forms of patterns** can be mined such as multilevel associations and multidimensional associations, quantitative association rules, rare patterns, and negative patterns. We can also mine high-dimensional patterns and compressed or approximate patterns.

- **Multilevel associations** involve data at more than one abstraction level (e.g., "*buys computer*" and "*buys laptop*"). These may be mined using multiple minimum support thresholds. **Multidimensional associations** contain more than one dimension. Techniques for mining such associations differ in how they handle repetitive predicates. **Quantitative association rules** involve quantitative attributes. Discretization, clustering, and statistical analysis that discloses exceptional behavior can be integrated with the pattern mining process.

- **Rare patterns** occur rarely but are of special interest. **Negative patterns** are patterns with components that exhibit negatively correlated behavior. Care should be taken in the definition of negative patterns, with consideration of the null-invariance property. Rare and negative patterns may highlight exceptional behavior in the data, which is likely of interest.

- **Constraint-based mining** strategies can be used to help direct the mining process toward patterns that match users' intuition or satisfy certain constraints. Many user-specified constraints can be pushed deep into the mining process. Constraints can be categorized into **pattern-pruning** and **data-pruning** constraints. Properties of such constraints include *monotonicity, antimonotonicity*, *data-antimonotonicity*, and *succinctness*. Constraints with such properties can be properly incorporated into efficient pattern mining processes.

- Methods have been developed for mining patterns in **high-dimensional space**. This includes a pattern growth approach based on *row enumeration* for mining data sets where the number of dimensions is large and the number of data tuples is small (e.g., for microarray data), as well as mining **colossal patterns** (i.e., patterns of very long length) by a *Pattern-Fusion* method.

- To reduce the number of patterns returned in mining, we can instead mine compressed patterns or approximate patterns. *Compressed patterns* can be mined with representative patterns defined based on the concept of clustering, and *approximate patterns* can be mined by extracting **redundancy-aware top-k patterns** (i.e., a small set of k-representative patterns that have not only high significance but also low redundancy with respect to one another).

- **Semantic annotations** can be generated to help users understand the meaning of the frequent patterns found, such as for textual terms like "{*frequent, pattern*}." These are dictionary-like annotations, providing semantic information relating to the term. This information consists of *context indicators* (e.g., terms indicating the context of that pattern), the most *representative data transactions* (e.g., fragments or sentences containing the term), and the most *semantically similar patterns* (e.g., "{*maximal, pattern*}" is semantically similar to "{*frequent, pattern*}"). The annotations provide a view of the pattern's context from different angles, which aids in their understanding.

- Frequent pattern mining has many diverse applications, ranging from pattern-based data cleaning to pattern-based classification, clustering, and outlier or exception analysis. These methods are discussed in the subsequent chapters in this book.

6.7 Exercises

1. Propose and outline a **level-shared mining** approach to mining multilevel association rules in which each item is encoded by its level position. Design it so that an initial scan of the database collects the count for each item *at each concept level*, identifying frequent and subfrequent items.

Comment on the processing cost of mining multilevel associations with this method in comparison to mining single-level associations.

2. Suppose, as manager of a chain of stores, you would like to use sales transactional data to analyze the effectiveness of your store's advertisements. In particular, you would like to study how specific factors influence the effectiveness of advertisements that announce a particular category of items on sale. The factors to study are the *region* in which customers live and the *day-of-the-week* and *time-of-the-day* of the ads. Discuss how to design an efficient method to mine the transaction data sets and explain how **multidimensional** and **multilevel mining** methods can help you derive a good solution.

3. **Quantitative association rules** may disclose exceptional behaviors within a data set, where "exceptional" can be defined based on statistical theory. For example, 6.1.3Section shows the association rule

   $$gender = female \Rightarrow mean\_wage = \$7.90/hr \ (overall\_mean\_wage = \$9.02/hr),$$

   which suggests an exceptional pattern. The rule states that the average wage for females is only $7.90 per hour, which is a significantly lower wage than the overall average of $9.02 per hour. Discuss how such quantitative rules can be discovered systematically and efficiently in large data sets with quantitative attributes.

4. In multidimensional data analysis, it is interesting to extract pairs of *similar* cell characteristics associated with substantial changes in measure in a data cube, where cells are considered *similar* if they are related by roll-up (i.e, *ancestors*), drill-down (i.e, *descendants*), or 1-D mutation (i.e, *siblings*) operations. Such an analysis is called **cube gradient analysis**.

   Suppose the measure of the cube is *average*. A user poses a set of *probe cells* and would like to find their corresponding sets of *gradient cells*, each of which satisfies a certain gradient threshold. For example, find the set of corresponding gradient cells that have an average sale price greater than 20% of that of the given probe cells. Develop an algorithm than mines the set of constrained gradient cells efficiently in a large data cube.

5. 6.1.5Section presented various ways of defining negatively correlated patterns. Consider 3Definition : "Suppose that itemsets X and Y are both frequent, that is, $sup(X) \geq min\_sup$ and $sup(Y) \geq min\_sup$, where min_sup is the minimum support threshold. If $(P(X|Y)+P(Y|X))/2 < \epsilon$, where ϵ is a negative pattern threshold, then pattern $X \cup Y$ is a **negatively correlated pattern**." Design an efficient pattern growth algorithm for mining the set of negatively correlated patterns.

6. Prove that each entry in the following table correctly characterizes its corresponding **rule constraint** for frequent itemset mining.

	Rule Constraint	*Antimonotonic*	*Monotonic*	*Succinct*
(a)	$v \in S$	no	yes	yes
(b)	$S \subseteq V$	yes	no	yes
(c)	$min(S) \leq v$	no	yes	yes
(d)	$range(S) \leq v$	yes	no	no
(e)	$variance(S) \leq v$	convertible	convertible	no

7. The price of each item in a store is non-negative. The store manager is only interested in rules of certain forms, using the constraints given in (a)–(b). For each of the following cases, identify the kinds of **constraints** they represent and briefly discuss how to mine such association rules using **constraint-based pattern mining**.

   (d) Containing at least one Blu-ray DVD movie.

   (d) Containing items with a sum of the prices that is less than $150.

   (d) Containing one free item and other items with a sum of the prices that is at least $200.

   (d) Where the average price of all the items is between $100 and $500.

8. **??**Section introduced a core Pattern-Fusion method for **mining high-dimensional data**. Explain why a long pattern, if one exists in the data set, is likely to be discovered by this method.

9. 6.2.1Section defined a **pattern distance measure** between closed patterns P_1 and P_2 as

$$Pat\_Dist(P_1, P_2) = 1 - \frac{|T(P_1) \cap T(P_2)|}{|T(P_1) \cup T(P_2)|},$$

   where $T(P_1)$ and $T(P_2)$ are the supporting transaction sets of P_1 and P_2, respectively. Is this a valid distance metric? Show the derivation to support your answer.

10. Association rule mining often generates a large number of rules, many of which may be similar, thus not containing much novel information. Design an efficient algorithm that **compresses** a large set of patterns into a small compact set. Discuss whether your mining method is robust under different pattern similarity definitions.

11. Frequent pattern mining may generate many superfluous patterns. Therefore, it is important to develop methods that mine compressed patterns. Suppose a user would like to obtain only k patterns (where k is a small integer). Outline an efficient method that generates the ***k* most representative patterns**, where more distinct patterns are preferred over very similar patterns. Illustrate the effectiveness of your method using a small data set.

12. It is interesting to generate **semantic annotations** for mined patterns. **??**Section presented a pattern annotation method. Alternative methods are possible, such as by utilizing type information. In the DBLP data set, for example, authors, conferences, terms, and papers form multi-typed data. Develop a method for automated semantic pattern annotation that makes good use of typed information.

6.8 Bibliographic Notes

This chapter described various ways in which the basic techniques of frequent itemset mining (presented in Chapter 6) have been extended. One line of extension is mining multilevel and multidimensional association rules. Multilevel association mining was studied in Srikant and Agrawal [SA95] and Han and Fu [HF95]. In Srikant and Agrawal [SA95], such mining was studied in the context of *generalized association rules*, and an R-interest measure was proposed for removing redundant rules. Mining multidimensional association rules using static discretization of quantitative attributes and data cubes was studied by Kamber, Han, and Chiang [KHC97].

Another line of extension is to mine patterns on numeric attributes. Srikant and Agrawal [SA96] proposed a nongrid-based technique for mining quantitative association rules, which uses a measure of partial completeness. Mining quantitative association rules based on rule clustering was proposed by Lent, Swami, and Widom [LSW97]. Techniques for mining quantitative rules based on x-monotone and rectilinear regions were presented by Fukuda, Morimoto, Morishita, and Tokuyama [FMMT96] and Yoda, Fukuda, Morimoto, et al. [YFM+97]. Mining (distance-based) association rules over interval data was proposed by Miller and Yang [MY97]. Aumann and Lindell [AL99] studied the mining of quantitative association rules based on a statistical theory to present only those rules that deviate substantially from normal data.

Mining rare patterns by pushing group-based constraints was proposed by Wang, He, and Han [WHH00]. Mining negative association rules was discussed by Savasere, Omiecinski, and Navathe [SON98] and by Tan, Steinbach, and Kumar [TSK05].

Constraint-based mining directs the mining process toward patterns that are likely of interest to the user. The use of metarules as syntactic or semantic filters defining the form of interesting single-dimensional association rules was proposed in Klemettinen, Mannila, Ronkainen, et al. [KMR+94]. Metarule-guided mining, where the metarule consequent specifies an action (e.g., Bayesian clustering or plotting) to be applied to the data satisfying the metarule antecedent, was proposed in Shen, Ong, Mitbander, and Zaniolo [SOMZ96]. A relation-based approach to metarule-guided mining of association rules was studied in Fu and Han [FH95].

Methods for constraint-based mining using pattern pruning constraints were studied by Ng, Lakshmanan, Han, and Pang [NLHP98]; Lakshmanan, Ng, Han, and Pang [LNHP99]; and Pei, Han, and Lakshmanan [PHL01]. Constraint-

based pattern mining by data reduction using data pruning constraints was studied by Bonchi, Giannotti, Mazzanti, and Pedreschi [BGMP03] and Zhu, Yan, Han, and Yu [ZYHY07]. An efficient method for mining constrained correlated sets was given in Grahne, Lakshmanan, and Wang [GLW00]. A dual mining approach was proposed by Bucila, Gehrke, Kifer, and White [BGKW03]. Other ideas involving the use of templates or predicate constraints in mining have been discussed in Anand and Kahn [AK93]; Dhar and Tuzhilin [DT93]; Hoschka and Klösgen [HK91]; Liu, Hsu, and Chen [LHC97]; Silberschatz and Tuzhilin [ST96]; and Srikant, Vu, and Agrawal [SVA97].

Traditional pattern mining methods encounter challenges when mining high-dimensional patterns, with applications like bioinformatics. Pan, Cong, Tung, et al. [PCT+03] proposed CARPENTER, a method for finding closed patterns in high-dimensional biological data sets, which integrates the advantages of vertical data formats and pattern growth methods. Pan, Tung, Cong, and Xu [PTCX04] proposed COBBLER, which finds frequent closed itemsets by integrating row enumeration with column enumeration. Liu, Han, Xin, and Shao [LHXS06] proposed TDClose to mine frequent closed patterns in high-dimensional data by starting from the maximal rowset, integrated with a row-enumeration tree. It uses the pruning power of the minimum support threshold to reduce the search space. For mining rather long patterns, called *colossal patterns*, Zhu, Yan, Han, et al. [ZYH+07] developed a core Pattern-Fusion method that leaps over an exponential number of intermediate patterns to reach colossal patterns.

To generate a reduced set of patterns, recent studies have focused on mining compressed sets of frequent patterns. Closed patterns can be viewed as a lossless compression of frequent patterns, whereas maximal patterns can be viewed as a simple lossy compression of frequent patterns. Top-k patterns, such as by Wang, Han, Lu, and Tsvetkov [WHLT05], and error-tolerant patterns, such as by Yang, Fayyad, and Bradley [YFB01], are alternative forms of interesting patterns. Afrati, Gionis, and Mannila [AGM04] proposed to use k-itemsets to cover a collection of frequent itemsets. For frequent itemset compression, Yan, Cheng, Han, and Xin [YCHX05] proposed a profile-based approach, and Xin, Han, Yan, and Cheng [XHYC05] proposed a clustering-based approach. By taking into consideration both pattern significance and pattern redundancy, Xin, Cheng, Yan, and Han [XCYH06] proposed a method for extracting redundancy-aware top-k patterns.

Automated semantic annotation of frequent patterns is useful for explaining the meaning of patterns. Mei, Xin, Cheng, et al. [MXC+07] studied methods for semantic annotation of frequent patterns.

An important extension to frequent itemset mining is mining sequence and structural data. This includes mining sequential patterns (Agrawal and Srikant [AS95]; Pei, Han, Mortazavi-Asl, et al. [PHMA+01, PHMA+04]; and Zaki [Zak01]); mining frequent espisodes (Mannila, Toivonen, and Verkamo [MTV97]); mining structural patterns (Inokuchi, Washio, and Motoda [IWM98]; Kuramochi and Karypis [KK01]; and Yan and Han [YH02]); mining cyclic association rules (Özden, Ramaswamy, and Silberschatz [ORS98]); intertransaction association rule mining (Lu, Han, and Feng [LHF98]); and calendric market basket analysis

(Ramaswamy, Mahajan, and Silberschatz [RMS98]). Mining such patterns is considered an advanced topic and readers are referred to these sources.

Pattern mining has been extended to help effective data classification and clustering. Pattern-based classification (Liu, Hsu, and Ma [LHM98] and Cheng, Yan, Han, and Hsu [CYHH07]) is discussed in Chapter 9. Pattern-based cluster analysis (Agrawal, Gehrke, Gunopulos, and Raghavan [AGGR98] and H. Wang, W. Wang, Yang, and Yu [WWYY02]) is discussed in Chapter 11.

Pattern mining also helps many other data analysis and processing tasks such as cube gradient mining and discriminative analysis (Imielinski, Khachiyan, and Abdulghani [IKA02]; Dong, Han, Lam, et al. [DHL+04]; Ji, Bailey, and Dong [JBD05]), discriminative pattern-based indexing (Yan, Yu, and Han [YYH05]), and discriminative pattern-based similarity search (Yan, Zhu, Yu, and Han [YZYH06]).

Pattern mining has been extended to mining spatial, temporal, time-series, and multimedia data, and data streams. Mining spatial association rules or spatial collocation rules was studied by Koperski and Han [KH95]; Xiong, Shekhar, Huang, et al. [XSH+04]; and Cao, Mamoulis, and Cheung [CMC05]. Pattern-based mining of time-series data is discussed in Shieh and Keogh [SK08] and Ye and Keogh [YK09]. There are many studies on pattern-based mining of multimedia data such as Zaïane, Han, and Zhu [ZHZ00] and Yuan, Wu, and Yang [YWY07]. Methods for mining frequent patterns on stream data have been proposed by many researchers, including Manku and Motwani [MM02]; Karp, Papadimitriou, and Shenker [KPS03]; and Metwally, Agrawal, and El Abbadi [MAA05]. These pattern mining methods are considered advanced topics.

Pattern mining has broad applications. Application areas include computer science such as software bug analysis, sensor network mining, and performance improvement of operating systems. For example, CP-Miner by Li, Lu, Myagmar, and Zhou [LLMZ04] uses pattern mining to identify copy-pasted code for bug isolation. PR-Miner by Li and Zhou [LZ05] uses pattern mining to extract application-specific programming rules from source code. Discriminative pattern mining is used for program failure detection to classify software behaviors (Lo, Cheng, Han, et al. [LCH+09]) and for troubleshooting in sensor networks (Khan, Le, Ahmadi, et al. [KLA+08]).

Bibliography

[AGGR98] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. In *Proc. 1998 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'98)*, pages 94–105, Seattle, WA, June 1998.

[AGM04] F. N. Afrati, A. Gionis, and H. Mannila. Approximating a collection of frequent sets. In *Proc. 2004 ACM SIGKDD Int. Conf. Knowledge Discovery in Databases (KDD'04)*, pages 12–19, Seattle, WA, Aug. 2004.

[AK93] T. Anand and G. Kahn. Opportunity explorer: Navigating large databases using knowledge discovery templates. In *Proc. AAAI-93 Workshop on Knowledge Discovery in Databases*, pages 45–51, Washington, DC, July 1993.

[AL99] Y. Aumann and Y. Lindell. A statistical theory for quantitative association rules. In *Proc. 1999 Int. Conf. Knowledge Discovery and Data Mining (KDD'99)*, pages 261–270, San Diego, CA, Aug. 1999.

[AS95] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proc. 1995 Int. Conf. Data Engineering (ICDE'95)*, pages 3–14, Taipei, Taiwan, Mar. 1995.

[BGKW03] C. Bucila, J. Gehrke, D. Kifer, and W. White. DualMiner: A dual-pruning algorithm for itemsets with constraints. *Data Mining and Knowledge Discovery*, 7:241–272, 2003.

[BGMP03] F. Bonchi, F. Giannotti, A. Mazzanti, and D. Pedreschi. ExAnte: Anticipated data reduction in constrained pattern mining. In *Proc. 7th European Conf. Principles and Pratice of Knowledge Discovery in Databases (PKDD'03)*, Sept. 2003.

[CMC05] H. Cao, N. Mamoulis, and D. W. Cheung. Mining frequent spatio-temporal sequential patterns. In *Proc. 2005 Int. Conf. Data Mining (ICDM'05)*, pages 82–89, Houston, TX, Nov. 2005.

[CYHH07] H. Cheng, X. Yan, J. Han, and C.-W. Hsu. Discriminative frequent pattern analysis for effective classification. In *Proc. 2007 Int. Conf. Data Engineering (ICDE'07)*, pages 716–725, Istanbul, Turkey, April 2007.

[DHL+04] G. Dong, J. Han, J. Lam, J. Pei, K. Wang, and W. Zou. Mining constrained gradients in multi-dimensional databases. *IEEE Trans. Knowledge and Data Engineering*, 16:922–938, 2004.

[DT93] V. Dhar and A. Tuzhilin. Abstract-driven pattern discovery in databases. *IEEE Trans. Knowledge and Data Engineering*, 5:926–938, 1993.

[FH95] Y. Fu and J. Han. Meta-rule-guided mining of association rules in relational databases. In *Proc. 1995 Int. Workshop on Integration of Knowledge Discovery with Deductive and Object-Oriented Databases (KDOOD'95)*, pages 39–46, Singapore, Dec. 1995.

[FMMT96] T. Fukuda, Y. Morimoto, S. Morishita, and T. Tokuyama. Data mining using two-dimensional optimized association rules: Scheme, algorithms, and visualization. In *Proc. 1996 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'96)*, pages 13–23, Montreal, Canada, June 1996.

[HF95] J. Han and Y. Fu. Discovery of multiple-level association rules from large databases. In *Proc. 1995 Int. Conf. Very Large Data Bases (VLDB'95)*, pages 420–431, Zurich, Switzerland, Sept. 1995.

[HK91] P. Hoschka and W. Klösgen. A support system for interpreting statistical data. In G. Piatetsky-Shapiro and W. J. Frawley, editors, *Knowledge Discovery in Databases*, pages 325–346. AAAI/MIT Press, 1991.

[IKA02] T. Imielinski, L. Khachiyan, and A. Abdulghani. Cubegrades: Generalizing association rules. *Data Mining and Knowledge Discovery*, 6:219–258, 2002.

[IWM98] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *Proc. 2000 European Symp. Principles of Data Mining and Knowledge Discovery (PKDD'00)*, pages 13–23, Lyon, France, Sept. 1998.

[JBD05] X. Ji, J. Bailey, and G. Dong. Mining minimal distinguishing subsequence patterns with gap constraints. In *Proc. 2005 Int. Conf. Data Mining (ICDM'05)*, pages 194–201, Houston, TX, Nov. 2005.

[KH95] K. Koperski and J. Han. Discovery of spatial association rules in geographic information databases. In *Proc. 1995 Int. Symp.*

Large Spatial Databases (SSD'95), pages 47–66, Portland, ME, Aug. 1995.

[KHC97] M. Kamber, J. Han, and J. Y. Chiang. Metarule-guided mining of multi-dimensional association rules using data cubes. In *Proc. 1997 Int. Conf. Knowledge Discovery and Data Mining (KDD'97)*, pages 207–210, Newport Beach, CA, Aug. 1997.

[KK01] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *Proc. 2001 Int. Conf. Data Mining (ICDM'01)*, pages 313–320, San Jose, CA, Nov. 2001.

[KLA+08] M. Khan, H. Le, H. Ahmadi, T. Abdelzaher, and J. Han. DustMiner: Troubleshooting interactive complexity bugs in sensor networks. In *Proc. 2008 ACM Int. Conf. Embedded Networked Sensor Systems (SenSys'08)*, Raleigh, NC, Nov. 2008.

[KMR+94] M. Klemettinen, H. Mannila, P. Ronkainen, H. Toivonen, and A. I. Verkamo. Finding interesting rules from large sets of discovered association rules. In *Proc. 3rd Int. Conf. Information and Knowledge Management*, pages 401–408, Gaithersburg, MD, Nov. 1994.

[KPS03] R. M. Karp, C. H. Papadimitriou, and S. Shenker. A simple algorithm for finding frequent elements in streams and bags. *ACM Trans. Database Systems*, 28, 2003.

[LCH+09] D. Lo, H. Cheng, J. Han, S. Khoo, and C. Sun. Classification of software behaviors for failure detection: A discriminative pattern mining approach. In *Proc. 2009 ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining (KDD'09)*, Paris, France, June 2009.

[LHC97] B. Liu, W. Hsu, and S. Chen. Using general impressions to analyze discovered classification rules. In *Proc. 1997 Int. Conf. Knowledge Discovery and Data Mining (KDD'97)*, pages 31–36, Newport Beach, CA, Aug. 1997.

[LHF98] H. Lu, J. Han, and L. Feng. Stock movement and n-dimensional inter-transaction association rules. In *Proc. 1998 SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery (DMKD'98)*, pages 12:1–12:7, Seattle, WA, June 1998.

[LHM98] B. Liu, W. Hsu, and Y. Ma. Integrating classification and association rule mining. In *Proc. 1998 Int. Conf. Knowledge Discovery and Data Mining (KDD'98)*, pages 80–86, New York, NY, Aug. 1998.

[LHXS06] H. Liu, J. Han, D. Xin, and Z. Shao. Mining frequent patterns on very high dimensional data: A top-down row enumeration approach. In *Proc. 2006 SIAM Int. Conf. Data Mining (SDM'06)*, Bethesda, MD, April 2006.

[LLMZ04] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *Proc. 2004 Symp. Operating Systems Design and Implementation (OSDI'04)*, San Francisco, CA, Dec. 2004.

[LNHP99] L.V.S. Lakshmanan, R. Ng, J. Han, and A. Pang. Optimization of constrained frequent set queries with 2-variable constraints. In *Proc. 1999 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'99)*, pages 157–168, Philadelphia, PA, June 1999.

[LSW97] B. Lent, A. Swami, and J. Widom. Clustering association rules. In *Proc. 1997 Int. Conf. Data Engineering (ICDE'97)*, pages 220–231, Birmingham, England, April 1997.

[LZ05] Z. Li and Y. Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *Proc. 2005 ACM SIGSOFT Symp. Foundations Software Eng. (FSE'05)*, Lisbon, Portugal, Sept. 2005.

[MAA05] A. Metwally, D. Agrawal, and A. El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *Proc. 2005 Int. Conf. Database Theory (ICDT'05)*, Edinburgh, UK, Jan. 2005.

[MM02] G. Manku and R. Motwani. Approximate frequency counts over data streams. In *Proc. 2002 Int. Conf. Very Large Data Bases (VLDB'02)*, pages 346–357, Hong Kong, China, Aug. 2002.

[MTV97] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, 1:259–289, 1997.

[MXC+07] Q. Mei, D. Xin, H. Cheng, J. Han, and C. Zhai. Semantic annotation of frequent patterns. *ACM Trans. Knowledge Discovery from Data (TKDD)*, 15:321–348, 2007.

[MY97] R. J. Miller and Y. Yang. Association rules over interval data. In *Proc. 1997 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'97)*, pages 452–461, Tucson, AZ, May 1997.

[NLHP98] R. Ng, L.V.S. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained associations rules. In *Proc. 1998 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'98)*, pages 13–24, Seattle, WA, June 1998.

[ORS98] B. Özden, S. Ramaswamy, and A. Silberschatz. Cyclic association rules. In *Proc. 1998 Int. Conf. Data Engineering (ICDE'98)*, pages 412–421, Orlando, FL, Feb. 1998.

[PCT+03] F. Pan, G. Cong, A.K.H. Tung, J. Yang, and M. Zaki. CARPENTER: Finding closed patterns in long biological datasets. In *Proc. 2003 ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining (KDD'03)*, pages 637–642, Washington, DC, Aug. 2003.

[PHL01] J. Pei, J. Han, and L.V.S. Lakshmanan. Mining frequent itemsets with convertible constraints. In *Proc. 2001 Int. Conf. Data Engineering (ICDE'01)*, pages 433–442, Heidelberg, Germany, April 2001.

[PHMA+01] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *Proc. 2001 Int. Conf. Data Engineering (ICDE'01)*, pages 215–224, Heidelberg, Germany, April 2001.

[PHMA+04] J. Pei, J. Han, B. Mortazavi-Asl, J. Wang, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. Mining sequential patterns by pattern-growth: The PrefixSpan approach. *IEEE Trans. Knowledge and Data Engineering*, 16:1424–1440, 2004.

[PTCX04] F. Pan, A.K.H. Tung, G. Cong, and X. Xu. COBBLER: Combining column and row enumeration for closed pattern discovery. In *Proc. 2004 Int. Conf. Scientific and Statistical Database Management (SSDBM'04)*, pages 21–30, Santorini Island, Greece, June 2004.

[RMS98] S. Ramaswamy, S. Mahajan, and A. Silberschatz. On the discovery of interesting patterns in association rules. In *Proc. 1998 Int. Conf. Very Large Data Bases (VLDB'98)*, pages 368–379, New York, NY, Aug. 1998.

[SA95] R. Srikant and R. Agrawal. Mining generalized association rules. In *Proc. 1995 Int. Conf. Very Large Data Bases (VLDB'95)*, pages 407–419, Zurich, Switzerland, Sept. 1995.

[SA96] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Proc. 5th Int. Conf. Extending Database Technology (EDBT'96)*, pages 3–17, Avignon, France, Mar. 1996.

[SK08] J. Shieh and E. Keogh. iSAX: Indexing and mining terabyte sized time series. In *Proc. 2008 ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining (KDD'08)*, Las Vegas, NV, Aug. 2008.

[SOMZ96] W. Shen, K. Ong, B. Mitbander, and C. Zaniolo. Metaqueries for data mining. In U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 375–398. AAAI/MIT Press, 1996.

[SON98] A. Savasere, E. Omiecinski, and S. Navathe. Mining for strong negative associations in a large database of customer transactions. In *Proc. 1998 Int. Conf. Data Engineering (ICDE'98)*, pages 494–502, Orlando, FL, Feb. 1998.

[ST96] A. Silberschatz and A. Tuzhilin. What makes patterns interesting in knowledge discovery systems. *IEEE Trans. Knowledge and Data Engineering*, 8:970–974, Dec. 1996.

[SVA97] R. Srikant, Q. Vu, and R. Agrawal. Mining association rules with item constraints. In *Proc. 1997 Int. Conf. Knowledge Discovery and Data Mining (KDD'97)*, pages 67–73, Newport Beach, CA, Aug. 1997.

[TSK05] P. N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*. Addison Wesley, 2005.

[VGS02] N. Vanetik, E. Gudes, and S. E. Shimony. Computing frequent graph patterns from semistructured data. In *Proc. 2002 Int. Conf. Data Mining (ICDM'02)*, pages 458–465, Maebashi, Japan, Dec. 2002.

[WHH00] K. Wang, Y. He, and J. Han. Mining frequent itemsets using support constraints. In *Proc. 2000 Int. Conf. Very Large Data Bases (VLDB'00)*, pages 43–52, Cairo, Egypt, Sept. 2000.

[WHLT05] J. Wang, J. Han, Y. Lu, and P. Tzvetkov. TFP: An efficient algorithm for mining top-k frequent closed itemsets. *IEEE Trans. Knowledge and Data Engineering*, 17:652–664, 2005.

[WWYY02] H. Wang, W. Wang, J. Yang, and P. S. Yu. Clustering by pattern similarity in large data sets. In *Proc. 2002 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'02)*, pages 418–427, Madison, WI, June 2002.

[XCYH06] D. Xin, H. Cheng, X. Yan, and J. Han. Extracting redundancy-aware top-k patterns. In *Proc. 2006 ACM SIGKDD Int. Conf. Knowledge Discovery in Databases (KDD'06)*, pages 444–453, Philadelphia, PA, Aug. 2006.

[XHYC05] D. Xin, J. Han, X. Yan, and H. Cheng. Mining compressed frequent-pattern sets. In *Proc. 2005 Int. Conf. Very Large Data Bases (VLDB'05)*, pages 709–720, Trondheim, Norway, Aug. 2005.

[XSH+04] H. Xiong, S. Shekhar, Y. Huang, V. Kumar, X. Ma, and J. S. Yoo. A framework for discovering co-location patterns in data sets with extended spatial objects. In *Proc. 2004 SIAM Int. Conf. Data Mining (SDM'04)*, Lake Buena Vista, FL, April 2004.

[YCHX05] X. Yan, H. Cheng, J. Han, and D. Xin. Summarizing itemset patterns: A profile-based approach. In *Proc. 2005 ACM SIGKDD Int. Conf. Knowledge Discovery in Databases (KDD'05)*, pages 314–323, Chicago, IL, Aug. 2005.

[YFB01] C. Yang, U. Fayyad, and P. S. Bradley. Efficient discovery of error-tolerant frequent itemsets in high dimensions. In *Proc. 2001 ACM SIGKDD Int. Conf. Knowledge Discovery in Databases (KDD'01)*, pages 194–203, San Fransisco, CA, Aug. 2001.

[YFM+97] K. Yoda, T. Fukuda, Y. Morimoto, S. Morishita, and T. Tokuyama. Computing optimized rectilinear regions for association rules. In *Proc. 1997 Int. Conf. Knowledge Discovery and Data Mining (KDD'97)*, pages 96–103, Newport Beach, CA, Aug. 1997.

[YH02] X. Yan and J. Han. gSpan: Graph-based substructure pattern mining. In *Proc. 2002 Int. Conf. Data Mining (ICDM'02)*, pages 721–724, Maebashi, Japan, Dec. 2002.

[YK09] L. Ye and E. Keogh. Time series shapelets: A new primitive for data mining. In *Proc. 2009 ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining (KDD'09)*, Paris, France, June 2009.

[YWY07] J. Yuan, Y. Wu, and M. Yang. Discovery of collocation patterns: From visual words to visual phrases. In *Proc. IEEE Conf. Computer Vision and Pattern Recognition (CVPR'07)*, Minneapolis, MN, June 2007.

[YYH05] X. Yan, P. S. Yu, and J. Han. Graph indexing based on discriminative frequent structure analysis. *ACM Trans. Database Systems*, 30:960–993, 2005.

[YZYH06] X. Yan, F. Zhu, P. S. Yu, and J. Han. Feature-based substructure similarity search. *ACM Trans. Database Systems*, 31:1418–1453, 2006.

[Zak01] M. Zaki. SPADE: An efficient algorithm for mining frequent sequences. *Machine Learning*, 40:31–60, 2001.

[ZHZ00] O. R. Zaïane, J. Han, and H. Zhu. Mining recurrent items in multimedia with progressive resolution refinement. In *Proc. 2000 Int. Conf. Data Engineering (ICDE'00)*, pages 461–470, San Diego, CA, Feb. 2000.

[ZYH+07] F. Zhu, X. Yan, J. Han, P. S. Yu, and H. Cheng. Mining colossal frequent patterns by core pattern fusion. In *Proc. 2007 Int. Conf. Data Engineering (ICDE'07)*, Istanbul, Turkey, April 2007.

[ZYHY07] F. Zhu, X. Yan, J. Han, and P. S. Yu. gPrune: A constraint pushing framework for graph pattern mining. In *Proc. 2007 Pacific-Asia Conf. Knowledge Discovery and Data Mining (PAKDD'07)*, Nanjing, China, May 2007.