Classification of large datasets has been proposed to be handled in several ways. Sampling has been proposed in [Cat91]. Partitioning the data and creating classifiers for each was proposed in [CS93]. Parallelization of classification algorithms has been examined [Fif92]. The SLIQ approach addresses scalability by a presorting step instead of sorting at each node, which some decision tree techniques require [MAR96]. The second improvement with SLIQ is that the tree is grown in a breadth-first fashion as opposed to a depth-first manner. In [SAM96], the authors not only propose SPRINT, but also design and compare several parallel implementations of SPRINT and SLIQ. Rainforest was proposed in 1998 [GRG98].

Although an RBF network could conceivably be of any shape, a seminal paper by Broomhead and Lowe proposed the structure discussed in this chapter [BL88]. An excellent and complete introduction to RBF networks is found in [Orr96]. The idea of the perceptron was proposed by Rosenblatt in [Ros58]. The MLP *is due to* Rumelhart and McClelland[RM86]. Discussions of choice for number of hidden layers in an MLP can be found in [Bis95] and [Hay99]. Kolmogorov's famous theorem is reported in [Kol57].

The 1R algorithm was studied in [Hol93] and [WF00]. PRISM was proposed in [Cen87]. Extracting rules from neural networks has been investigated since the early 1990s [Liu95], [TS93], [LSL95], and [LSL96].

CHAPTER 5

Clustering

5.1 INTRODUCTION
5.2 SIMILARITY AND DISTANCE MEASURES
5.3 OUTLIERS
5.4 HIERARCHICAL ALGORITHMS
5.5 PARTITIONAL ALGORITHMS
5.6 CLUSTERING LARGE DATABASES
5.7 CLUSTERING WITH CATEGORICAL ATTRIBUTES
5.8 COMPARISON
5.9 EXERCISES
5.10 BIBLIOGRAPHIC NOTES

5.1 INTRODUCTION

Clustering is similar to classification in that data are grouped. However, unlike classification, the groups are not predefined. Instead, the grouping is accomplished by finding similarities between data according to characteristics found in the actual data. The groups are called *clusters*. Some authors view clustering as a special type of classification. In this text, however, we follow a more conventional view in that the two are different. Many definitions for clusters have been proposed:

- Set of like elements. Elements from different clusters are not alike.
- The distance between points in a cluster is less than the distance between a point in the cluster and any point outside it.

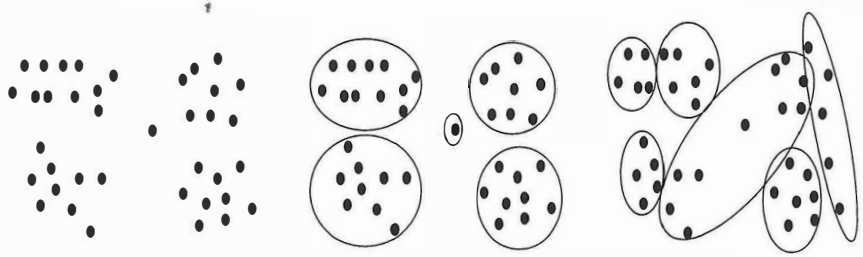
A term similar to clustering is *database segmentation*, where like tuples (records) in a database are grouped together. This is done to partition or segment the database into components that then give the user a more general view of the data. In this text, we do not differentiate between segmentation and clustering. A simple example of clustering is found in Example 5.1. This example illustrates the fact that determining how to do the clustering is not straightforward.

EXAMPLE 5.1

An international online catalog company wishes to group its customers based on common features. Company management does not have any predefined labels for these groups. Based on the outcome of the grouping, they will target marketing and advertising campaigns to the different groups. The information they have about the customers includes

TABLE 5.1: Sample Data for Example 5.1

Income	Age	Children	Marital Status	Education
$25,000	35	3	Single	High school
$15,000	25	1	Married	High school
$20,000	40	0	Single	High school
$30,000	20	0	Divorced	High school
$20,000	25	3	Divorced	College
$70,000	60	0	Married	College
$90,000	30	0	Married	Graduate school
$200,000	45	5	Married	Graduate school
$100,000	50	2	Divorced	College



(a) Group of homes (b) Geographic distance-based (c) Size-based

FIGURE 5.1: Different clustering attributes.

income, age, number of children, marital status, and education. Table 5 shows some tuples from this database for customers in the United States. Depending on the type of advertising, not all attributes are important. For example, suppose the advertising is for a special sale on children's clothes. We could target the advertising only to the persons with children. One possible clustering is that shown by the divisions of the table. The first group of people have young children and a high school degree, while the second group is similar but have no children. The third group has both children and a college degree. The last two groups have higher incomes and at least a college degree. The very last group has children. Different clusterings would have been found by examining age or marital status.

As illustrated in Figure 5.1, a given set of data may be clustered on different attributes. Here a group of homes in a geographic area is shown. The first type of clustering is based on the location of the home. Homes that are geographically close to each other are clustered together. In the second clustering, homes are grouped based on the size of the house.

Clustering has been used in many application domains, including biology, medicine, anthropology, marketing, and economics. Clustering applications include plant and animal

classification, disease classification, image processing, pattern recognition, and document retrieval. One of the first domains in which clustering was used was biological taxonomy. Recent uses include examining Web log data to detect usage patterns.

When clustering is applied to a real-world database, many interesting problems occur:

- Outlier handling is difficult. Here the elements do not naturally fall into any cluster. They can be viewed as solitary clusters. However, if a clustering algorithm attempts to find larger clusters, these outliers will be forced to be placed in some cluster. This process may result in the creation of poor clusters by combining two existing clusters and leaving the outlier in its own cluster.
- Dynamic data in the database implies that cluster membership may change over time.
- Interpreting the semantic meaning of each cluster may be difficult. With classification, the labeling of the classes is known ahead of time. However, with clustering, this may not be the case. Thus, when the clustering process finishes creating a set of clusters, the exact meaning of each cluster may not be obvious. Here is where a domain expert is needed to assign a label or interpretation for each cluster.
- There is no one correct answer to a clustering problem. In fact, many answers may be found. The exact number of clusters required is not easy to determine. Again, a domain expert may be required. For example, suppose we have a set of data about plants that have been collected during a field trip. Without any prior knowledge of plant classification, if we attempt to divide this set of data into similar groupings, it would not be clear how many groups should be created.
- Another related issue is what data should be used for clustering. Unlike learning during a classification process, where there is some a priori knowledge concerning what the attributes of each classification should be, in clustering we have no supervised learning to aid the process. Indeed, clustering can be viewed as similar to unsupervised learning.

We can then summarize some basic features of clustering (as opposed to classification):

- The (best) number of clusters is not known.
- There may not be any a priori knowledge concerning the clusters.
- Cluster results are dynamic.

The clustering problem is stated as shown in Definition 5.1. Here we assume that the number of clusters to be created is an input value, k. The actual content (and interpretation) of each cluster, K_j, $1 \leq j \leq k$, is determined as a result of the function definition. Without loss of generality, we will view that the result of solving a clustering problem is that a set of clusters is created: $K = \{K_1, K_2, \ldots, K_k\}$.

DEFINITION 5.1. Given a database $D = \{t_1, t_2, \ldots, t_n\}$ of tuples and an integer value k, the **clustering problem** is to define a mapping $f : D \rightarrow \{1, \ldots, k\}$ where each t_i is assigned to one cluster K_j, $1 \leq j \leq k$. A **cluster**, K_j, contains precisely those tuples mapped to it; that is, $K_j = \{t_i \mid f(t_i) = K_j, 1 \leq i \leq n, \text{ and } t_i \in D\}$.

A classification of the different types of clustering algorithms is shown in Figure 5.2. Clustering algorithms themselves may be viewed as hierarchical or partitional. With

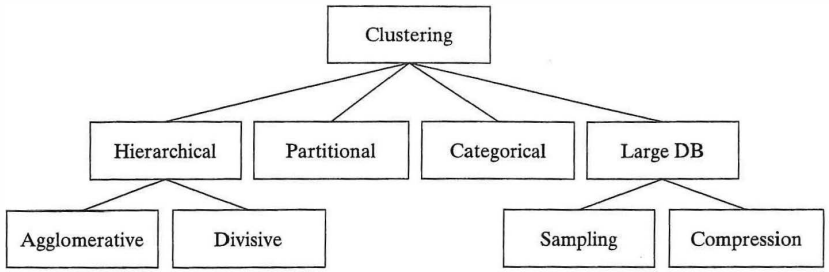


FIGURE 5.2: Classification of clustering algorithms.

hierarchical clustering, a nested set of clusters is created. Each level in the hierarchy has a separate set of clusters. At the lowest level, each item is in its own unique cluster. At the highest level, all items belong to the same cluster. With hierarchical clustering, the desired number of clusters is not input. With *partitional* clustering, the algorithm creates only one set of clusters. These approaches use the desired number of clusters to drive how the final set is created. Traditional clustering algorithms tend to be targeted to small numeric databases that fit into memory. There are, however, more recent clustering algorithms that look at categorical data and are targeted to larger, perhaps dynamic, databases. Algorithms targeted to larger databases may adapt to memory constraints by either sampling the database or using data structures, which can be compressed or pruned to fit into memory regardless of the size of the database. Clustering algorithms may also differ based on whether they produce overlapping or nonoverlapping clusters. Even though we consider only nonoverlapping clusters, it is possible to place an item in multiple clusters. In turn, nonoverlapping clusters can be viewed as extrinsic or intrinsic. *Extrinsic* techniques use labeling of the items to assist in the classification process. These algorithms are the traditional classification supervised learning algorithms in which a special input training set is used. *Intrinsic* algorithms do not use any a priori category labels, but depend only on the adjacency matrix containing the distance between objects. All algorithms we examine in this chapter fall into the intrinsic class.

The types of clustering algorithms can be furthered classified based on the implementation technique used. Hierarchical algorithms can be categorized as agglomerative or divisive. "*Agglomerative*" implies that the clusters are created in a bottom-up fashion, while *divisive* algorithms work in a top-down fashion. Although both hierarchical and partitional algorithms could be described using the agglomerative vs. divisive label, it typically is more associated with hierarchical algorithms. Another descriptive tag indicates whether each individual element is handled one by one, *serial* (sometimes called *incremental*), or whether all items are examined together, *simultaneous*. If a specific tuple is viewed as having attribute values for all attributes in the schema, then clustering algorithms could differ as to how the attribute values are examined. As is usually done with decision tree classification techniques, some algorithms examine attribute values one at a time, *monothetic*. *Polythetic* algorithms consider all attribute values at one time. Finally, clustering algorithms can be labeled based on the mathematical formulation given to the algorithm: graph theoretic or matrix algebra. In this chapter we generally use the graph approach and describe the input to the clustering algorithm as an adjacency matrix labeled with distance measures.

We discuss many clustering algorithms in the following sections. This is only a representative subset of the many algorithms that have been proposed in the literature. Before looking at these algorithms, we first examine possible similarity measures and examine the impact of outliers.

5.2 SIMILARITY AND DISTANCE MEASURES

There are many desirable properties for the clusters created by a solution to a specific clustering problem. The most important one is that a tuple within one cluster is more like tuples within that cluster than it is similar to tuples outside it. As with classification, then, we assume the definition of a similarity measure, $\text{sim}(t_i, t_l)$, defined between any two tuples, $t_i, t_l \in D$. This provides a more strict and alternative clustering definition, as found in Definition 5.2. Unless otherwise stated, we use the first definition rather than the second. Keep in mind that the similarity relationship stated within the second definition is a desirable, although not always obtainable, property.

DEFINITION 5.2. Given a database $D = \{t_1, t_2, \ldots, t_n\}$ of tuples, a similarity measure, $\text{sim}(t_i, t_l)$, defined between any two tuples, $t_i, t_l \in D$, and an integer value k, the **clustering problem** is to define a mapping $f : D \rightarrow \{1, \ldots, k\}$ where each t_i is assigned to one cluster K_j, $1 \leq j \leq k$. Given a cluster, K_j, $\forall t_{jl}, t_{jm} \in K_j$ and $t_i \notin K_j$, $\text{sim}(t_{jl}, t_{jm}) > \text{sim}(t_{jl}, t_i)$.

A distance measure, $\text{dis}(t_i, t_j)$, as opposed to similarity, is often used in clustering. The clustering problem then has the desirable property that given a cluster, K_j, $\forall t_{jl}$, $t_{jm} \in K_j$ and $t_i \notin K_j$, $\text{dis}(t_{jl}, t_{jm}) \leq \text{dis}(t_{jl}, t_i)$.

Some clustering algorithms look only at numeric data, usually assuming metric data points. *Metric* attributes satisfy the triangular inequality. The clusters can then be described by using several characteristic values. Given a cluster, K_m of N points $\{t_{m1}, t_{m2}, \ldots, t_{mN}\}$, we make the following definitions [ZRL96]:

$$\text{centroid} = C_m = \frac{\sum_{i=1}^{N}(t_{mi})}{N} \tag{5.1}$$

$$\text{radius} = R_m = \sqrt{\frac{\sum_{i=1}^{N}(t_{mi} - C_m)^2}{N}} \tag{5.2}$$

$$\text{diameter} = D_m = \sqrt{\frac{\sum_{i=1}^{N}\sum_{j=1}^{N}(t_{mi} - t_{mj})^2}{(N)(N-1)}} \tag{5.3}$$

Here the centroid is the "middle" of the cluster; it need not be an actual point in the cluster. Some clustering algorithms alternatively assume that the cluster is represented by one centrally located object in the cluster called a *medoid*. The radius is the square root of the average mean squared distance from any point in the cluster to the centroid, and the diameter is the square root of the average mean squared distance between all pairs of points in the cluster. We use the notation M_m to indicate the medoid for cluster K_m.

Many clustering algorithms require that the distance between clusters (rather than elements) be determined. This is not an easy task given that there are many interpretations for distance between clusters. Given clusters K_i and K_j, there are several standard alternatives to calculate the distance between clusters. A representative list is:

- **Single link:** Smallest distance between an element in one cluster and an element in the other. We thus have $\text{dis}(K_i, K_j) = \min(\text{dis}(t_{il}, t_{jm}))\forall t_{il} \in K_i \notin K_j$ and $\forall t_{jm} \in K_j \notin K_i$.
- **Complete link:** Largest distance between an element in one cluster and an element in the other. We thus have $\text{dis}(K_i, K_j) = \max(\text{dis}(t_{il}, t_{jm}))\forall t_{il} \in K_i \notin K_j$ and $\forall t_{jm} \in K_j \notin K_i$.
- **Average:** Average distance between an element in one cluster and an element in the other. We thus have $\text{dis}(K_i, K_j) = \text{mean}(\text{dis}(t_{il}, t_{jm}))\forall t_{il} \in K_i \notin K_j$ and $\forall t_{jm} \in K_j \notin K_i$.
- **Centroid:** If clusters have a representative centroid, then the centroid distance is defined as the distance between the centroids. We thus have $\text{dis}(K_i, K_j) = \text{dis}(C_i, C_j)$, where C_i is the centroid for K_i and similarly for C_j.
- **Medoid:** Using a medoid to represent each cluster, the distance between the clusters can be defined by the distance between the medoids: $\text{dis}(K_i, K_j) = \text{dis}(M_i, M_j)$.

5.3 OUTLIERS

As mentioned earlier, *outliers* are sample points with values much different from those of the remaining set of data. Outliers may represent errors in the data (perhaps a malfunctioning sensor recorded an incorrect data value) or could be correct data values that are simply much different from the remaining data. A person who is 2.5 meters tall is much taller than most people. In analyzing the height of individuals, this value probably would be viewed as an outlier.

Some clustering techniques do not perform well with the presence of outliers. This problem is illustrated in Figure 5.3. Here if three clusters are found (solid line), the outlier will occur in a cluster by itself. However, if two clusters are found (dashed line), the two (obviously) different sets of data will be placed in one cluster because they are closer together than the outlier. This problem is complicated by the fact that many clustering algorithms actually have as input the number of desired clusters to be found.

Clustering algorithms may actually find and remove outliers to ensure that they perform better. However, care must be taken in actually removing outliers. For example, suppose that the data mining problem is to predict flooding. Extremely high water level values occur very infrequently, and when compared with the normal water level values may seem to be outliers. However, removing these values may not allow the data mining algorithms to work effectively because there would be no data that showed that floods ever actually occurred.

Outlier detection, or *outlier mining*, is the process of identifying outliers in a set of data. Clustering, or other data mining, algorithms may then choose to remove or treat these values differently. Some outlier detection techniques are based on statistical techniques. These usually assume that the set of data follows a known distribution and that outliers can be detected by well-known tests such as *discordancy tests*. However, these

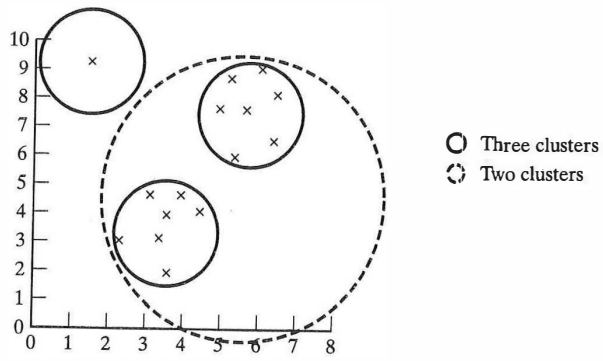


FIGURE 5.3: Outlier clustering problem.

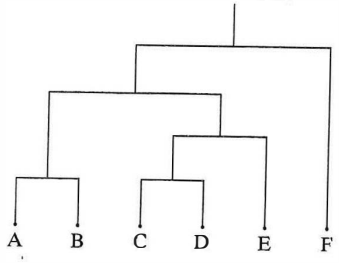


FIGURE 5.4: Dendrogram for Example 5.2.

tests are not very realistic for real-world data because real-world data values may not follow well-defined data distributions. Also, most of these tests assume a single attribute value, and many attributes are involved in real-world datasets. Alternative detection techniques may be based on distance measures.

5.4 HIERARCHICAL ALGORITHMS

As mentioned earlier, hierarchical clustering algorithms actually creates sets of clusters. Example 5.2 illustrates the concept. Hierarchical algorithms differ in how the sets are created. A tree data structure, called a *dendrogram*, can be used to illustrate the hierarchical clustering technique and the sets of different clusters. The root in a dendrogram tree contains one cluster where all elements are together. The leaves in the dendrogram each consist of a single element cluster. Internal nodes in the dendrogram represent new clusters formed by merging the clusters that appear as its children in the tree. Each level in the tree is associated with the distance measure that was used to merge the clusters. All clusters created at a particular level were combined because the children clusters had a distance between them less than the distance value associated with this level in the tree. A dendrogram for Example 5.2 is seen in Figure 5.4.

EXAMPLE 5.2

Figure 5.5 shows six elements, {A, B, C, D, E, F}, to be clustered. Parts (a) to (e) of the figure show five different sets of clusters. In part (a) each cluster is viewed to consist of

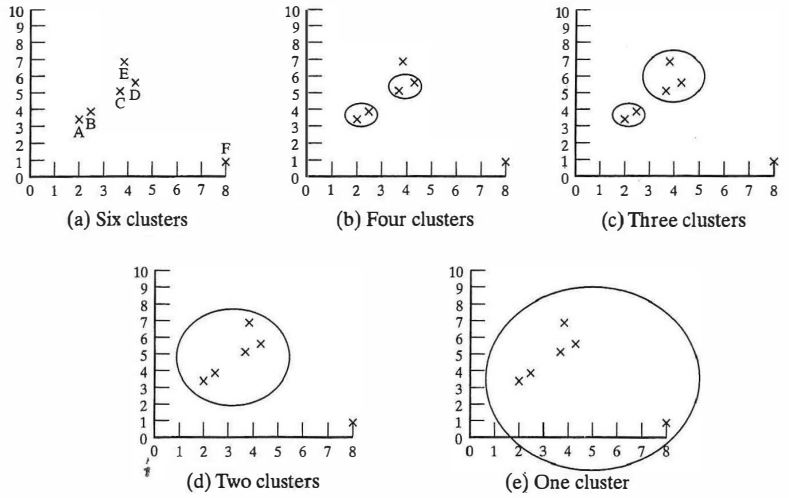


FIGURE 5.5: Five levels of clustering for Example 5.2.

a single element. Part (b) illustrates four clusters. Here there are two sets of two-element clusters. These clusters are formed at this level because these two elements are closer to each other than any of the other elements. Part (c) shows a new cluster formed by adding a close element to one of the two-element clusters. In part (d) the two-element and three-element clusters are merged to give a five-element cluster. This is done because these two clusters are closer to each other than to the remote element cluster, {F}. At the last stage, part (e), all six elements are merged.

The space complexity for hierarchical algorithms is $O(n^2)$ because this is the space required for the adjacency matrix. The space required for the dendrogram is $O(kn)$, which is much less than $O(n^2)$. The time complexity for hierarchical algorithms is $O(kn^2)$ because there is one iteration for each level in the dendrogram. Depending on the specific algorithm, however, this could actually be $O(maxd\ n^2)$ where *maxd* is the maximum distance between points. Different algorithms may actually merge the closest clusters from the next lowest level or simply create new clusters at each level with progressively larger distances.

Hierarchical techniques are well suited for many clustering applications that naturally exhibit a nesting relationship between clusters. For example, in biology, plant and animal taxonomies could easily be viewed as a hierarchy of clusters.

5.4.1 Agglomerative Algorithms

Agglomerative algorithms start with each individual item in its own cluster and iteratively merge clusters until all items belong in one cluster. Different agglomerative algorithms differ in how the clusters are merged at each level. Algorithm 5.1 illustrates the typical agglomerative clustering algorithm. It assumes that a set of elements and distances between them is given as input. We use an $n \times n$ vertex adjacency matrix, A, as input.

Here the adjacency matrix, A, contains a distance value rather than a simple boolean value: $A[i, j] = \text{dis}(t_i, t_j)$. The output of the algorithm is a dendrogram, DE, which we represent as a set of ordered triples $\langle d, k, K \rangle$ where d is the threshold distance, k is the number of clusters, and K is the set of clusters. The dendrogram in Figure 5.7(a) would be represented by the following:

$$\{\langle 0, 5, \{\{A\}, \{B\}, \{C\}, \{D\}, \{E\}\}\rangle, \langle 1, 3, \{\{A, B\}, \{C, D\}, \{E\}\}\rangle$$
$$\langle 2, 2, \{\{A, B, C, D\}, \{E\}\}\rangle, \langle 3, 1, \{\{A, B, C, D, E\}\}\rangle\}$$

Outputting the dendrogram produces a set of clusters rather than just one clustering. The user can determine which of the clusters (based on distance threshold) he or she wishes to use.

ALGORITHM 5.1

```
Input:
   D = {t1, t2, ..., tn}    //Set of elements
   A      //Adjacency matrix showing distance between elements
Output:
   DE    // Dendrogram represented as a set of ordered triples
Agglomerative algorithm:
   d = 0;
   k = n;
   K = {{t1}, ..., {tn}};
   DE = {⟨d, k, K⟩}; // Initially dendrogram contains each element
                     in its own cluster.
   repeat
      oldk = k;
      d = d + 1;
      Ad = Vertex adjacency matrix for graph with threshold
           distance of d;
      ⟨k, K⟩ = NewClusters(Ad, D);
      if oldk ≠ k then
         DE = DE ∪ ⟨d, k, K⟩; // New set of clusters added to dendrogram.
         until k = 1
```

This algorithm uses a procedure called *NewClusters* to determine how to create the next level of clusters from the previous level. This is where the different types of agglomerative algorithms differ. It is possible that only two clusters from the prior level are merged or that multiple clusters are merged. Algorithms also differ in terms of which clusters are merged when there are several clusters with identical distances. In addition, the technique used to determine the distance between clusters may vary. *Single link*, *complete link*, and *average link* techniques are perhaps the most well known agglomerative techniques based on well-known graph theory concepts.

All agglomerative approaches experience excessive time and space constraints. The space required for the adjacency matrix is $O(n^2)$ where there are n items to cluster. Because of the iterative nature of the algorithm, the matrix (or a subset of it) must be accessed multiple times. The simplistic algorithm provided in Algorithm 5.1 performs at most *maxd* examinations of this matrix, where *maxd* is the largest distance between any two points. In addition, the complexity of the *NewClusters* procedure could be expensive. This is a potentially severe problem in large databases. Another issue with

the agglomerative approach is that it is not incremental. Thus, when new elements are added or old ones are removed or changed, the entire algorithm must be rerun. More recent incremental variations, as discussed later in this text, address this problem.

Single Link Technique. The single link technique is based on the idea of finding maximal connected components in a graph. A *connected component* is a graph in which there exists a path between any two vertices. With the single link approach, two clusters are merged if there is at least one edge that connects the two clusters; that is, if the minimum distance between any two points is less than or equal to the threshold distance being considered. For this reason, it is often called the *nearest neighbor* clustering technique. Example 5.3 illustrates this process.

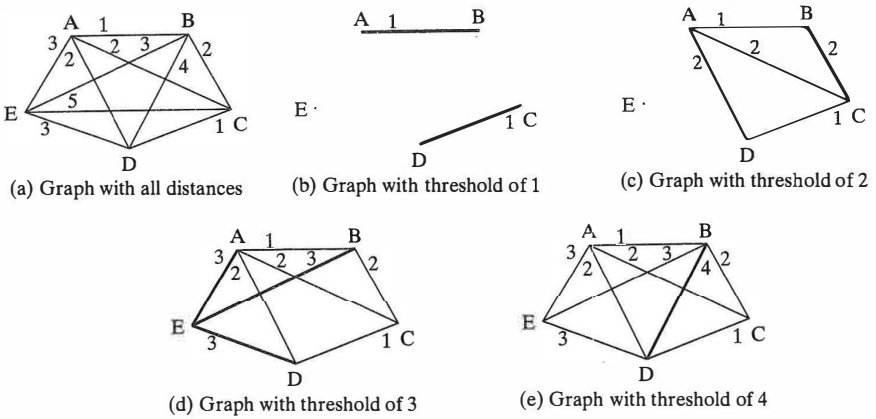
EXAMPLE 5.3

Table 5.2 contains five sample data items with the distance between the elements indicated in the table entries. When viewed as a graph problem, Figure 5.6(a) shows the general graph with all edges labeled with the respective distances. To understand the idea behind the hierarchical approach, we show several graph variations in Figures 5.6(b), (c), (d), and (e). Figure 5.6(b) shows only those edges with a distance of 1 or less. There are only two edges. The first level of single link clustering then will combine the connected clusters (single elements from the first phase), giving three clusters: {A,B}, {C,D}, and {E}. During the next level of clustering, we look at edges with a length of 2 or less. The graph representing this threshold distance is shown in Figure 5.6(c). Note that we now have an edge (actually three) between the two clusters {A,B} and {C,D}. Thus, at this level of the single link clustering algorithm, we merge these two clusters to obtain a total of two clusters: {A,B,C,D} and {E}. The graph that is created with a threshold distance of 3 is shown in Figure 5.6(d). Here the graph is connected, so the two clusters from the last level are merged into one large cluster that contains all elements. The dendrogram for this single link example is shown in Figure 5.7(a). The labeling on the right-hand side shows the threshold distance used to merge the clusters at each level.

The single link algorithm is obtained by replacing the *NewClusters* procedure in the agglomerative algorithm with a procedure to find connected components of a graph. We assume that this connected components procedure has as input a graph (actually represented by a vertex adjacency matrix and set of vertices) and as outputs a set of

TABLE 5.2: Sample Data for Example 5.3

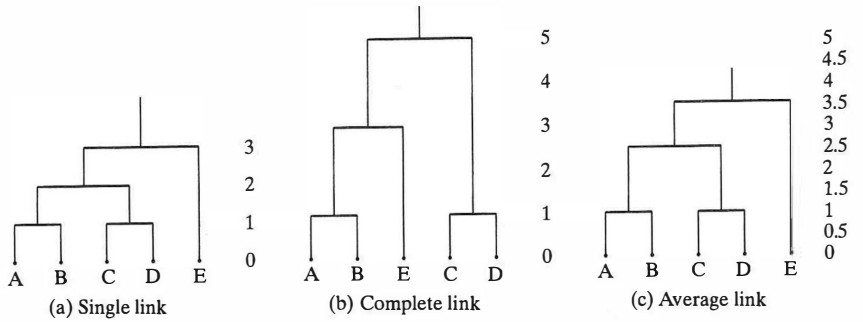
Item	A	B	C	D	E
A	0	1	2	2	3
B	1	0	2	4	3
C	2	2	0	1	5
D	2	4	1	0	3
E	3	3	5	3	0



(a) Graph with all distances (b) Graph with threshold of 1 (c) Graph with threshold of 2

(d) Graph with threshold of 3 (e) Graph with threshold of 4

FIGURE 5.6: Graphs for Example 5.3.



(a) Single link (b) Complete link (c) Average link

FIGURE 5.7: Dendrograms for Example 5.3.

connected components defined by a number (indicating the number of components) and an array containing the membership of each component. Note that this is exactly what the last two entries in the ordered triple are used for by the dendrogram data structure.

The single link approach is quite simple, but it suffers from several problems. This algorithm is not very efficient because the connected components procedure, which is an $O(n^2)$ space and time algorithm, is called at each iteration. A more efficient algorithm could be developed by looking at which clusters from an earlier level can be merged at each step. Another problem is that the clustering creates clusters with long chains.

An alternative view to merging clusters in the single link approach is that two clusters are merged at a stage where the threshold distance is d if the minimum distance between any vertex in one cluster and any vertex in the other cluster is at most d.

There have been other variations of the single link algorithm. One variation, based on the use of a *minimum spanning tree (MST)*, is shown in Algorithm 5.2. Here we assume that a procedure, *MST*, produces a minimum spanning tree given an adjacency matrix as input. The clusters are merged in increasing order of the distance found in the MST. In the algorithm we show that once two clusters are merged, the distance between

them in the tree becomes ∞. Alternatively, we could have replaced the two nodes and edge with one node.

ALGORITHM 5.2

```
Input:
   D = {t1, t2, ..., tn}   //Set of elements
   A      //Adjacency matrix showing distance between elements
Output:
   DE     // Dendrogram represented as a set of ordered triples
MST single link algorithm:
   d = 0
   k = n
   K = {{t1}, ..., {tn}}
   DE = (d, k, K); // Initially dendrogram contains each element in
         its own cluster.
   M = MST(A);
   repeat
      oldk = k;
      Ki, Kj = two clusters closest together in MST;
      K = K - {Ki} - {Kj} U {Ki U Kj};
      k = oldk - 1;
      d = dis(Ki, Kj);
      DE = DE U (d, k, K); // New set of clusters added to dendrogram.
      dis(Ki, Kj) = ∞
   until k = 1
```

We illustrate this algorithm using the data in Example 5.3. Figure 5.8 shows one MST for the example. The algorithm will merge A and B and then C and D (or the reverse). These two clusters will then be merged at a threshold of 2. Finally, E will be merged at a threshold of 3. Note that we get exactly the same dendrogram as in Figure 5.7(a).

The time complexity of this algorithm is $O(n^2)$ because the procedure to create the minimum spanning tree is $O(n^2)$ and it dominates the time of the algorithm. Once it is created having $n - 1$ edges, the *repeat* loop will be repeated only $n - 1$ times.

The single linkage approach is infamous for its chain effect; that is, two clusters are merged if only two of their points are close to each other. There may be points in the respective clusters to be merged that are far apart, but this has no impact on the algorithm. Thus, resulting clusters may have points that are not related to each other at all, but simply happen to be near (perhaps via a transitive relationship) points that are close to each other.

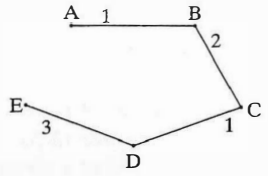


FIGURE 5.8: MST for Example 5.3.

Complete Link Algorithm. Although the complete link algorithm is similar to the single link algorithm, it looks for cliques rather than connected components. A *clique* is a maximal graph in which there is an edge between any two vertices. Here a procedure is used to find the maximum distance between any clusters so that two clusters are merged if the maximum distance is less than or equal to the distance threshold. In this algorithm, we assume the existence of a procedure, clique, which finds all cliques in a graph. As with the single link algorithm, this is expensive because it is an $O(n^2)$ algorithm.

Clusters found with the complete link method tend to be more compact than those found using the single link technique. Using the data found in Example 5.3, Figure 5.7(b) shows the dendrogram created. A variation of the complete link algorithm is called the *farthest neighbor algorithm*. Here the closest clusters are merged where the distance is the smallest measured by looking at the maximum distance between any two points.

Average Link. The average link technique merges two clusters if the average distance between any two points in the two target clusters is below the distance threshold. The algorithm used here is slightly different from that found in single and complete link algorithms because we must examine the complete graph (not just the threshold graph) at each stage. Thus, we restate this algorithm in Algorithm 5.3.

ALGORITHM 5.3

```
Input:
   D = {t1, t2, ..., tn}   //Set of elements
   A      //Adjacency matrix showing distance between elements
Output:
   DE     // Dendrogram represented as a set of ordered triples
Average link algorithm:
   d = 0;
   k = n;
   K = {{t1}, ..., {tn}};
   DE = (d, k, K); // Initially dendrogram contains each element
                   in its own cluster.
   repeat
      oldk = k;
      d = d + 0.5;
      for each pair of Ki, Kj ∈ K do
         ave = average distance between all ti ∈ Ki and tj ∈ Kj;
         if ave ≤ d, then
            K = K - {Ki} - {Kj} U {Ki U Kj};
            k = oldk - 1;
            DE = DE U (d, k, K); // New set of clusters added
                                    to dendrogram.
      until k = 1
```

Note that in this algorithm we increment d by 0.5 rather than by 1. This is a rather arbitrary decision based on understanding of the data. Certainly, we could have used an increment of 1, but we would have had a dendrogram different from that seen in Figure 5.7(c).

5.4.2 Divisive Clustering

With divisive clustering, all items are initially placed in one cluster and clusters are repeatedly split in two until all items are in their own cluster. The idea is to split up clusters where some elements are not sufficiently close to other elements.

One simple example of a divisive algorithm is based on the MST version of the single link algorithm. Here, however, we cut out edges from the MST from the largest to the smallest. Looking at Figure 5.8, we would start with a cluster containing all items: $\{A, B, C, D, E\}$. Looking at the MST, we see that the largest edge is between D and E. Cutting this out of the MST, we then split the one cluster into two: $\{E\}$ and $\{A, B, C, D\}$. Next we remove the edge between B and C. This splits the one large cluster into two: $\{A, B\}$ and $\{C, D\}$. These will then be split at the next step. The order depends on how a specific implementation would treat identical values. Looking at the dendrogram in Figure 5.7(a), we see that we have created the same set of clusters as with the agglomerative approach, but in reverse order.

5.5 PARTITIONAL ALGORITHMS

Nonhierarchical or partitional clustering creates the clusters in one step as opposed to several steps. Only one set of clusters is created, although several different sets of clusters may be created internally within the various algorithms. Since only one set of clusters is output, the user must input the desired number, k, of clusters. In addition, some metric or criterion function is used to determine the goodness of any proposed solution. This measure of quality could be the average distance between clusters or some other metric. The solution with the best value for the criterion function is the clustering solution used. One common measure is a squared error metric, which measures the squared distance from each point to the centroid for the associated cluster:

$$\sum_{m=1}^{k} \sum_{t_{mi} \in K_m} \text{dis}(C_m, t_{mi})^2 \tag{5.4}$$

A problem with partitional algorithms is that they suffer from a combinatorial explosion due to the number of possible solutions. Clearly, searching all possible clustering alternatives usually would not be feasible. For example, given a measurement criteria, a naive approach could look at all possible sets of k clusters. There are $S(n, k)$ possible combinations to examine. Here

$$S(n, k) = \frac{1}{k!} \sum_{i=1}^{k} (-1)^{k-i} \binom{k}{i} (i)^n \tag{5.5}$$

There are 11,259,666,000 different ways to cluster 19 items into 4 clusters. Thus, most algorithms look only at a small subset of all the clusters using some strategy to identify sensible clusters. Because of the plethora of partitional algorithms, we will look at only a representative few. We have chosen some of the most well known algorithms as well as some others that have appeared recently in the literature.

5.5.1 Minimum Spanning Tree

Since we have agglomerative and divisive algorithms based on the use of an MST, we also present a partitional MST algorithm. This is a very simplistic approach, but it illustrates how partitional algorithms work. The algorithm is shown in Algorithm 5.4. Since the clustering problem is to define a mapping, the output of this algorithm shows the clusters as a set of ordered pairs $\langle t_i, j \rangle$ where $f(t_i) = K_j$.

ALGORITHM 5.4

```
Input:
   D = {t1, t2, ..., tn}   //Set of elements
   A     //Adjacency matrix showing distance between elements
   k     //Number of desired clusters
Output:
   f     //Mapping represented as a set of ordered pairs
Partitional MST algorithm:
   M = MST(A)
   identify inconsistent edges in M;
   remove k - 1 inconsistent edges;
   create output representation;
```

The problem is how to define "inconsistent." It could be defined as in the earlier division MST algorithm based on distance. This would remove the largest $k-1$ edges from the starting completely connected graph and yield the same results as this corresponding level in the dendrogram. Zahn proposes more reasonable inconsistent measures based on the weight (distance) of an edge as compared to those close to it. For example, an inconsistent edge would be one whose weight is much larger than the average of the adjacent edges.

The time complexity of this algorithm is again dominated by the *MST* procedure, which is $O(n^2)$. At most, $k-1$ edges will be removed, so the last three steps of the algorithm, assuming each step takes a constant time, is only $O(k-1)$. Although determining the inconsistent edges in M may be quite complicated, it will not require a time greater than the number of edges in M. When looking at edges adjacent to one edge, there are at most $k-2$ of these edges. In this case, then, the last three steps are $O(k^2)$, and the total algorithm is still $O(n^2)$.

5.5.2 Squared Error Clustering Algorithm

The *squared error* clustering algorithm minimizes the squared error. The *squared error for a cluster* is the sum of the squared Euclidean distances between each element in the cluster and the cluster centroid, C_k. Given a cluster K_i, let the set of items mapped to that cluster be $\{t_{i1}, t_{i2}, \ldots, t_{im}\}$. The squared error is defined as

$$se_{K_i} = \sum_{j=1}^{m} \|t_{ij} - C_k\|^2 \tag{5.6}$$

Given a set of clusters $K = \{K_1, K_2, \ldots, K_k\}$, the *squared error* for K is defined as

$$se_K = \sum_{j=1}^{k} se_{K_j} \tag{5.7}$$

In actuality, there are many different examples of squared error clustering algorithms. They all follow the basic algorithm structure shown in Algorithm 5.5.

ALGORITHM 5.5

```
Input:
   D = {t1, t2, ..., tn}   //Set of elements
   k      //Number of desired clusters
Output:
   K      //Set of clusters
Squared error algorithm:
   assign each item ti to a cluster;
   calculate center for each cluster;
   repeat
      assign each item ti to the cluster which has the closest center;
      calculate new center for each cluster;
      calculate squared error;
   until the difference between successive squared errors
      is below a threshold;
```

For each iteration in the squared error algorithm, each tuple is assigned to the cluster with the closest center. Since there are k clusters and n items, this is an $O(kn)$ operation. Assuming t iterations, this becomes an $O(tkn)$ algorithm. The amount of space may be only $O(n)$ because an adjacency matrix is not needed, as the distance between all items is not used.

5.5.3 *K*-Means Clustering

K-means is an iterative clustering algorithm in which items are moved among sets of clusters until the desired set is reached. As such, it may be viewed as a type of squared error algorithm, although the convergence criteria need not be defined based on the squared error. A high degree of similarity among elements in clusters is obtained, while a high degree of dissimilarity among elements in different clusters is achieved simultaneously. The *cluster mean* of $K_i = \{t_{i1}, t_{i2}, \ldots, t_{im}\}$ is defined as

$$m_i = \frac{1}{m} \sum_{j=1}^{m} t_{ij} \tag{5.8}$$

This definition assumes that each tuple has only one numeric value as opposed to a tuple with many attribute values. The K-means algorithm requires that some definition of cluster mean exists, but it does not have to be this particular one. Here the mean is defined identically to our earlier definition of centroid. This algorithm assumes that the desired number of clusters, k, is an input parameter. Algorithm 5.6 shows the K-means algorithm. Note that the initial values for the means are arbitrarily assigned. These could be assigned randomly or perhaps could use the values from the first k input items themselves. The convergence criteria could be based on the squared error, but they need not be. For example, the algorithm could stop when no (or a very small) number of tuples are assigned to different clusters. Other termination techniques have simply looked at a fixed number of iterations. A maximum number of iterations may be included to ensure stopping even without convergence.

ALGORITHM 5.6

```
Input:
   D = {t1, t2, ..., tn}   //Set of elements
   k      //Number of desired clusters
Output:
   K      //Set of clusters
K-means algorithm:
   assign initial values for means m1, m2, ..., mk;
   repeat
      assign each item ti to the cluster which has the closest mean;
      calculate new mean for each cluster;
   until convergence criteria is met;
```

The K-means algorithm is illustrated in Example 5.4.

EXAMPLE 5.4

Suppose that we are given the following items to cluster:

$$\{2, 4, 10, 12, 3, 20, 30, 11, 25\} \tag{5.9}$$

and suppose that $k = 2$. We initially assign the means to the first two values: $m_1 = 2$ and $m_2 = 4$. Using Euclidean distance, we find that initially $K_1 = \{2, 3\}$ and $K_2 = \{4, 10, 12, 20, 30, 11, 25\}$. The value 3 is equally close to both means, so we arbitrarily choose K_1. Any desired assignment could be used in the case of ties. We then recalculate the means to get $m_1 = 2.5$ and $m_2 = 16$. We again make assignments to clusters to get $K_1 = \{2, 3, 4\}$ and $K_2 = \{10, 12, 20, 30, 11, 25\}$. Continuing in this fashion, we obtain the following:

m_1	m_2	K_1	K_2
3	18	{2, 3, 4, 10}	{12, 20, 30, 11, 25}
4.75	19.6	{2, 3, 4, 10, 11, 12}	{20, 30, 25}
7	25	{2, 3, 4, 10, 11, 12}	{20, 30, 25}

Note that the clusters in the last two steps are identical. This will yield identical means, and thus the means have converged. Our answer is thus $K_1 = \{2, 3, 4, 10, 11, 12\}$ and $K_2 = \{20, 30, 25\}$.

The time complexity of K-means is $O(tkn)$ where t is the number of iterations. K-means finds a local optimum and may actually miss the global optimum. K-means does not work on categorical data because the mean must be defined on the attribute type. Only convex-shaped clusters are found. It also does not handle outliers well. One variation of K-means, *K-modes*, does handle categorical data. Instead of using means, it uses modes. A typical value for k is 2 to 10.

Although the K-means algorithm often produces good results, it is not time-efficient and does not scale well. By saving distance information from one iteration to the next, the actual number of distance calculations that must be made can be reduced.

Some K-means variations examine ways to improve the chances of finding the global optimum. This often involves careful selection of the initial clusters and means. Another variation is to allow clusters to be split and merged. The variance within a cluster is examined, and if it is too large, a cluster is split. Similarly, if the distance between two cluster centroids is less than a predefined threshold, they will be combined.

5.5.4 Nearest Neighbor Algorithm

An algorithm similar to the single link technique is called the *nearest neighbor algorithm*. With this serial algorithm, items are iteratively merged into the existing clusters that are closest. In this algorithm a threshold, t, is used to determine if items will be added to existing clusters or if a new cluster is created.

ALGORITHM 5.7

```
Input:
   D = {t1, t2, ..., tn}   //Set of elements
   A      //Adjacency matrix showing distance between elements
Output:
   K      //Set of clusters
Nearest neighbor algorithm:
   K1 = {t1};
   K = {K1};
   k = 1;
   for i = 2 to n do
      find the tm in some cluster Km in K such that dis(ti, tm) is
         the smallest;
      if dis(ti, tm), ≤ t then
         Km = Km ∪ ti
      else
         k = k + 1;
         Kk = {ti};
```

Example 5.5 shows the application of the nearest neighbor algorithm to the data shown in Table 5.2 assuming a threshold of 2. Notice that the results are the same as those seen in Figure 5.7(a) at the level of 2.

EXAMPLE 5.5

Initially, A is placed in a cluster by itself, so we have $K_1 = \{A\}$. We then look at B to decide if it should be added to K_1 or be placed in a new cluster. Since $\text{dis}(A, B) = 1$, which is less than the threshold of 2, we place B in K_1 to get $K_1 = \{A, B\}$. When looking at C, we see that its distance to both A and B is 2, so we add it to the cluster to get $K_1 = \{A, B, C\}$. The $\text{dis}(D, C) = 1 < 2$, so we get $K_1 = \{A, B, C, D\}$. Finally, looking at E, we see that the closest item in K_1 has a distance of 3, which is greater than 2, so we place it in its own cluster: $K_2 = \{E\}$.

The complexity of the nearest neighbor algorithm actually depends on the number of items. For each loop, each item must be compared to each item already in a cluster. Obviously, this is n in the worst case. Thus, the time complexity is $O(n^2)$. Since we do need to examine the distances between items often, we assume that the space requirement is also $O(n^2)$.

5.5.5 PAM Algorithm

The *PAM (partitioning around medoids)* algorithm, also called the *K-medoids* algorithm, represents a cluster by a medoid. Using a medoid is an approach that handles outliers well. The PAM algorithm is shown in Algorithm 5.8. Initially, a random set of k items is taken to be the set of medoids. Then at each step, all items from the input dataset that are not currently medoids are examined one by one to see if they should be medoids. That is, the algorithm determines whether there is an item that should replace one of the existing medoids. By looking at all pairs of medoid, non-medoid objects, the algorithm chooses the pair that improves the overall quality of the clustering the best and exchanges them. Quality here is measured by the sum of all distances from a non-medoid object to the medoid for the cluster it is in. An item is assigned to the cluster represented by the medoid to which it is closest (minimum distance). We assume that K_i is the cluster represented by medoid t_i. Suppose t_i is a current medoid and we wish to determine whether it should be exchanged with a non-medoid t_h. We wish to do this swap only if the overall impact to the cost (sum of the distances to cluster medoids) represents an improvement.

Following the lead in [NH94], we use C_{jih} to be the cost change for an item t_j associated with swapping medoid t_i with non-medoid t_h. The cost is the change to the sum of all distances from items to their cluster medoids. There are four cases that must be examined when calculating this cost:

1. $t_j \in K_i$, but $\exists$ another medoid t_m where $\text{dis}(t_j, t_m) \leq \text{dis}(t_j, t_h)$;
2. $t_j \in K_i$, but $\text{dis}(t_j, t_h) \leq \text{dis}(t_j, t_m) \forall$ other medoids t_m;
3. $t_j \in K_m, \notin K_i$, and $\text{dis}(t_j, t_m) \leq \text{dis}(t_j, t_h)$; and
4. $t_j \in K_m, \notin K_i$, but $\text{dis}(t_j, t_h) \leq \text{dis}(t_j, t_m)$.

We leave it as an exercise to determine the cost of each of these cases. The total impact to quality by a medoid change TC_{ih} then is given by

$$TC_{ih} = \sum_{j=1}^{n} C_{jih} \tag{5.10}$$

ALGORITHM 5.8

```
Input:
   D = {t1, t2, ..., tn}   //Set of elements
   A      //Adjacency matrix showing distance between elements
   k      //Number of desired clusters
Output:
   K      //Set of clusters
PAM algorithm:
   arbitrarily select k medoids from D;
   repeat
      for each th not a medoid do
         for each medoid ti do
            calculate TCih;
      find i, h where TCih is the smallest;
      if TCih < 0, then
         replace medoid ti with th;
   until TCih ≥ 0;
   for each ti ∈ D do
      assign ti to Kj, where dis(ti, tj) is the smallest over all medoids;
```

Example 5.6 shows the application of the PAM algorithm to the data shown in Table 5.2 assuming a threshold of 2.

EXAMPLE 5.6

Suppose that the two medoids that are initially chosen are A and B. Based on the distances shown in Table 5.2 and randomly placing items when distances are identical to the two medoids, we obtain the clusters $\{A, C, D\}$ and $\{B, E\}$ The three non-medoids, $\{C, D, E\}$, are then examined to see which (if any) should be used to replace A or B. We thus have six costs to determine: TC_{AC}, TC_{AD}, TC_{AE}, TC_{BC}, TC_{BD}, and TC_{BE}. Here we use the name of the item instead of a numeric subscript value. We obtain the following:

$$TC_{AC} = C_{AAC} + C_{BAC} + C_{CAC} + C_{DAC} + C_{EAC} = 1 + 0 - 2 - 1 + 0 = -2 \quad (5.11)$$

Here A is no longer a medoid, and since it is closer to B, it will be placed in the cluster with B as medoid, and thus its cost is $C_{AAC} = 1$. The cost for B is 0 because it stays a cluster medoid. C is now a medoid, so it has a negative cost based on its distance to the old medoid; that is, $C_{CAB} = -2$. D is closer to C than it was to A by a distance of 1, so its cost is $C_{DAC} = -1$. Finally, E stays in the same cluster with the same distance, so its cost change is 0. Thus, we have that the overall cost is a reduction of 2. Figure 5.9 illustrates the calculation of these six costs. Looking at these, we see that the minimum cost is 2 and that there are several ways to reduce this cost. Arbitrarily choosing the first swap, we get C and B as the new medoids with the clusters being $\{C, D\}$ and $\{B, A, E\}$. This concludes the first iteration of PAM. At the next iteration, we examine changing medoids again and pick the choice that best reduces the cost. The iterations stop when no changes will reduce the cost. We leave the rest of this problem to the reader as an exercise.

PAM does not scale well to large datasets because of its computational complexity. For each iteration, we have $k(n-k)$ pairs of objects i, h for which a cost, TC_{ih}, should be determined. Calculating the cost during each iteration requires that the cost be calculated for all other non-medoids t_j. There are $n - k$ of these. Thus, the total complexity per iteration is $n(n-k)^2$. The total number of iterations can be quite large, so PAM is not an alternative for large databases. However, there are some clustering algorithms based on PAM that are targeted to large datasets.

CLARA (Clustering LARge Applications) improves on the time complexity of PAM by using samples of the dataset. The basic idea is that it applies PAM to a sample of the underlying database and then uses the medoids found as the medoids for the complete clustering. Each item from the complete database is then assigned to the cluster with the medoid to which it is closest. To improve the CLARA accuracy, several samples can be drawn with PAM applied to each. The sample chosen as the final clustering is the one that performs the best. Because of the sampling, CLARA is more efficient than PAM for large databases. However, it may not be as effective, depending on the sample size. Five samples of size $40 + 2k$ seem to give good results [KR90].

CLARANS (clustering large applications based upon randomized search) improves on CLARA by using multiple different samples. In addition to the normal input to PAM,

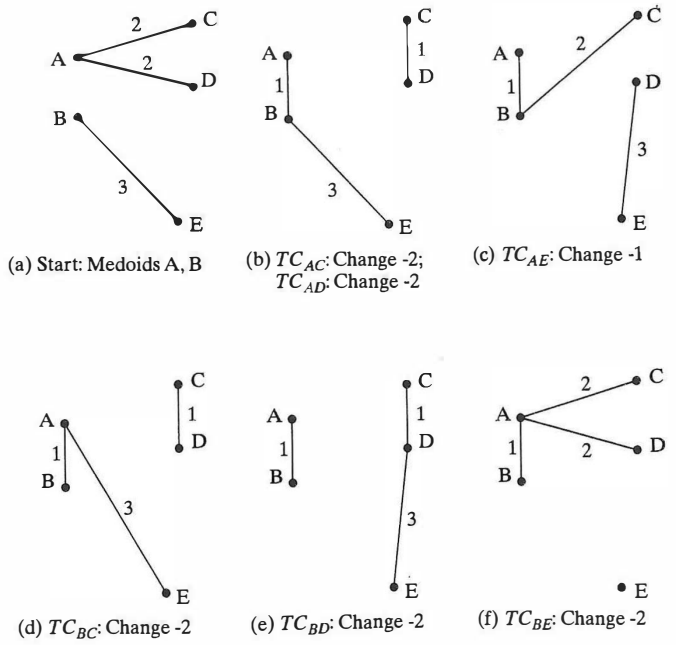


FIGURE 5.9: Cost calculations for Example 5.6.

CLARANS requires two additional parameters: *maxneighbor* and *numlocal*. *Maxneighbor* is the number of neighbors of a node to which any specific node can be compared. As *maxneighbor* increases, CLARANS looks more and more like PAM because all nodes will be examined. *Numlocal* indicates the number of samples to be taken. Since a new clustering is performed on each sample, this also indicates the number of clusterings to be made. Performance studies indicate that *numlocal* = 2 and *maxneighbor* = max((0.0125 × $k(n-k)$), 250) are good choices [NH94]. CLARANS is shown to be more efficient than either PAM or CLARA for any size dataset. CLARANS assumes that all data are in main memory. This certainly is not a valid assumption for large databases.

5.5.6 Bond Energy Algorithm

The *bond energy algorithm (BEA)* was developed and has been used in the database design area to determine how to group data and how to physically place data on a disk. It can be used to cluster attributes based on usage and then perform logical or physical design accordingly. With BEA, the *affinity* (bond) between database attributes is based on common usage. This bond is used by the clustering algorithm as a similarity measure. The actual measure counts the number of times the two attributes are used together in a given time. To find this, all common queries must be identified.

The idea is that attributes that are used together form a cluster and should be stored together. In a distributed database, each resulting cluster is called a *vertical fragment* and

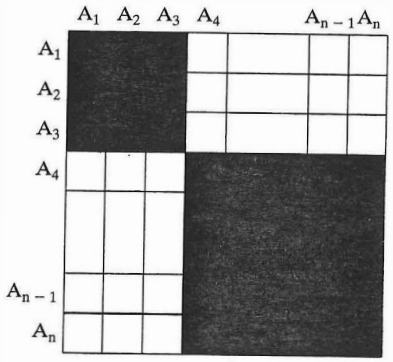


FIGURE 5.10: Clustered affinity matrix for BEA (modified from [ÖV99]).

may be stored at different sites from other fragments. The basic steps of this clustering algorithm are:

1. Create an attribute affinity matrix in which each entry indicates the affinity between the two associate attributes. The entries in the similarity matrix are based on the frequency of common usage of attribute pairs.
2. The BEA then converts this similarity matrix to a BOND matrix in which the entries represent a type of nearest neighbor bonding based on probability of co-access. The BEA algorithm rearranges rows or columns so that similar attributes appear close together in the matrix.
3. Finally, the designer draws boxes around regions in the matrix with high similarity.

The resulting matrix, modified from [ÖV99], is illustrated in Figure 5.10 The two shaded boxes represent the attributes that have been grouped together into two clusters.

Two attributes A_i and A_j have a high affinity if they are frequently used together in database applications. At the heart of the BEA algorithm is the global affinity measure. Suppose that a database schema consists of n attributes $\{A_1, A_2, \ldots, A_n\}$. The global affinity measure, AM, is defined as

$$AM = \sum_{i=1}^{n} (\text{bond}(A_i, A_{i-1}) + \text{bond}(A_i, A_{i+1})) \tag{5.12}$$

5.5.7 Clustering with Genetic Algorithms

There have been clustering techniques based on the use of genetic algorithms. To determine how to perform clustering with genetic algorithms, we first must determine how to represent each cluster. One simple approach would be to use a bit-map representation for each possible cluster. So, given a database with four items, $\{A, B, C, D\}$, we would represent one solution to creating two clusters as 1001 and 0110. This represents the two clusters $\{A, D\}$ and $\{B, C\}$.

Algorithm 5.9 shows one possible iterative refinement technique for clustering that uses a genetic algorithm. The approach is similar to that in the squared error approach in that an initial random solution is given and successive changes to this converge on a local optimum. A new solution is generated from the previous solution using crossover and mutation operations. Our algorithm shows only crossover. The use of crossover to create a new solution from a previous solution is shown in Example 5.7. The new "solution" must be created in such a way that it represents a valid k clustering. A fitness function must be used and may be defined based on an inverse of the squared error. Because of the manner in which crossover works, genetic clustering algorithms perform a global search rather than a local search of potential solutions.

ALGORITHM 5.9

```
Input:
   D = {t1, t2, ..., tn}   //Set of elements
   k      //Number of desired clusters
Output:
   K      //Set of clusters
GA clustering algorithm:
   randomly create an initial solution;
   repeat
      use crossover to create a new solution;
   until termination criteria is met;
```

EXAMPLE 5.7

Suppose a database contains the following eight items $\{A, B, C, D, E, F, G, H\}$, which are to be placed into three clusters. We could initially place the items into the three clusters $\{A, C, E\}$, $\{B, F\}$, and $\{D, G, H\}$, which are represented by 10101000, 01000100, and 00010011, respectively. Suppose we choose the first and third individuals as parents and do a simple crossover at point 4. This yields the new solution: 00011000, 01000100, and 10100011.

5.5.8 Clustering with Neural Networks

Neural networks (NNs) that use unsupervised learning attempt to find features in the data that characterize the desired output. They look for clusters of like data. These types of NNs are often called *self-organizing neural networks*. There are two basic types of unsupervised learning: noncompetitive and competitive.

With the *noncompetitive* or *Hebbian* learning, the weight between two nodes is changed to be proportional to both output values. That is

$$\Delta w_{ji} = \eta y_j y_i \tag{5.13}$$

With *competitive* learning, nodes are allowed to compete and the winner takes all. This approach usually assumes a two-layer NN in which all nodes from one layer are connected to all nodes in the other layer. As training occurs, nodes in the output layer become associated with certain tuples in the input dataset. Thus, this provides a grouping

of these tuples together into a cluster. Imagine every input tuple having each attribute value input to a specific input node in the NN. The number of input nodes is the same as the number of attributes. We can thus associate each weight to each output node with one of the attributes from the input tuple. When a tuple is input to the NN, all output nodes produce an output value. The node with the weights more similar to the input tuple is declared the winner. Its weights are then adjusted. This process continues with each tuple input from the training set. With a large and varied enough training set, over time each output node should become associated with a set of tuples. The input weights to the node are then close to an average of the tuples in this cluster.

Self-Organizing Feature Maps. A *self-organizing feature map (SOFM)* or *self-organizing map (SOM)* is an NN approach that uses competitive unsupervised learning. Learning is based on the concept that the behavior of a node should impact only those nodes and arcs near it. Weights are initially assigned randomly and adjusted during the learning process to produce better results. During this learning process, hidden features or patterns in the data are uncovered and the weights are adjusted accordingly. SOFMs were developed by observing how neurons work in the brain and in ANNs. That is [BS97]:

- The firing of neurons impact the firing of other neurons that are near it.
- Neurons that are far apart seem to inhibit each other.
- Neurons seem to have specific nonoverlapping tasks.

The term *self-organizing* indicates the ability of these NNs to organize the nodes into clusters based on the similarity between them. Those nodes that are closer together are more similar than those that are far apart. This hints at how the actual clustering is performed. Over time, nodes in the output layer become matched to input nodes, and patterns of nodes in the output layer emerge.

Perhaps the most common example of a SOFM is the *Kohonen self-organizing map*, which is used extensively in commercial data mining products to perform clustering. There is one input layer and one special layer, which produces output values that compete. In effect, multiple outputs are created and the best one is chosen. This extra layer is not technically either a hidden layer or an output layer, so we refer to it here as the *competitive layer*. Nodes in this layer are viewed as a two-dimensional grid of nodes as seen in Figure 5.11. Each input node is connected to each node in this grid. Propagation occurs by sending the input value for each input node to each node in the competitive layer. As with regular NNs, each arc has an associated weight and each node in the competitive layer has an activation function. Thus, each node in the competitive layer produces an output value, and the node with the best output wins the competition and is determined to be the output for that input. An attractive feature of Kohonen nets is that the data can be fed into the multiple competitive nodes in parallel. Training occurs by adjusting weights so that the best output is even better the next time this input is used. "Best" is determined by computing a distance measure.

A common approach is to initialize the weights on the input arcs to the competitive layer with normalized values. The similarity between output nodes and input vectors is then determined by the dot product of the two vectors. Given an input tuple $X = \langle x_1, \ldots, x_h \rangle$ and weights on arcs input to a competitive node i as $w_{1i}, \ldots, w_{hi}$, the

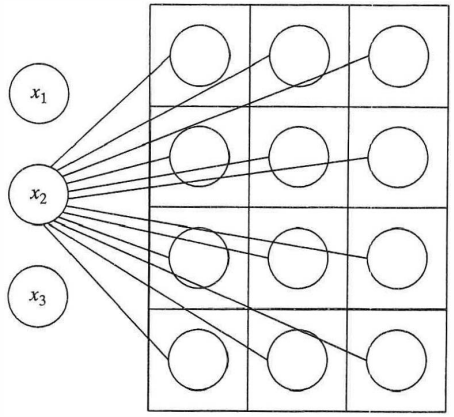


FIGURE 5.11: Kohonen network.

similarity between X and i can be calculated by

$$\text{sim}(X, i) = \sum_{j=1}^{h} x_j \, w_{ji} \tag{5.14}$$

The competitive node most similar to the input node wins the competitive. Based on this, the weights coming into i as well as those for the nodes immediately surrounding it in the matrix are increased. This is the learning phase. Given a node i, we use the notation N_i to represent the union of i and the nodes near it in the matrix. Thus, the learning process uses

$$\Delta w_{kj} = \left\{ \begin{array}{ll} c(x_k - w_{kj}) & \text{if } j \in N_i \\ 0 & \text{otherwise} \end{array} \right\} \tag{5.15}$$

In this formula, c indicates the learning rate and may actually vary based on the node rather than being a constant. The basic idea of SOM learning is that after each input tuple in the training set, the winner and its neighbors have their weights changed to be closer to that of the tuple. Over time, a pattern on the output nodes emerges, which is close to that of the training data. At the beginning of the training process, the neighborhood of a node may be defined to be large. However, the neighborhood may decrease during the processing.

5.6 CLUSTERING LARGE DATABASES

The clustering algorithms presented in the preceding sections are some of the classic clustering techniques. When clustering is used with dynamic databases, these algorithms may not be appropriate. First, they all assume that [because most are $O(n^2)$] sufficient main memory exists to hold the data to be clustered and the data structures needed to support them. With large databases containing thousands of items (or more), these assumptions are not realistic. In addition, performing I/Os continuously through the multiple iterations of an algorithm is too expensive. Because of these main memory restrictions, the algorithms do not scale up to large databases. Another issue is that some assume that the

data are present all at once. These techniques are not appropriate for dynamic databases. Clustering techniques should be able to adapt as the database changes.

The algorithms discussed in the following subsections each examine an issue associated with performing clustering in a database environment. It has been argued that to perform effectively on large databases, a clustering algorithm should [BFR98]:

1. require no more (preferably less) than one scan of the database.
2. have the ability to provide status and "best" answer so far during the algorithm execution. This is sometimes referred to as the ability to be *online.*
3. be suspendable, stoppable, and resumable.
4. be able to update the results incrementally as data are added or removed from the database.
5. work with limited main memory.
6. be capable of performing different techniques for scanning the database. This may include sampling.
7. process each tuple only once.

Recent research at Microsoft has examined how to efficiently perform the clustering algorithms with large databases [BFR98]. The basic idea of this scaling approach is as follows:

1. Read a subset of the database into main memory.
2. Apply clustering technique to data in memory.
3. Combine results with those from prior samples.
4. The in-memory data are then divided into three different types: those items that will always be needed even when the next sample is brought in, those that can be discarded with appropriate updates to data being kept in order to answer the problem, and those that will be saved in a compressed format. Based on the type, each data item is then kept, deleted, or compressed in memory.
5. If termination criteria are not met, then repeat from step 1.

This approach has been applied to the K-means algorithm and has been shown to be effective.

5.6.1 BIRCH

BIRCH (balanced iterative reducing and clustering using hierarchies) is designed for clustering a large amount of metric data. It assumes that there may be a limited amount of main memory and achieves a linear I/O time requiring only one database scan. It is incremental and hierarchical, and it uses an outlier handling technique. Here points that are found in sparsely populated areas are removed. The basic idea of the algorithm is that a tree is built that captures needed information to perform clustering. The clustering is then performed on the tree itself, where labelings of nodes in the tree contain the needed information to calculate distance values. A major characteristic of the BIRCH algorithm is the use of the *clustering feature*, which is a triple that contains information about a cluster (see Definition 5.3). The clustering feature provides a summary of the information about one cluster. By this definition it is clear that BIRCH applies only to numeric data.

This algorithm uses a tree called a *CF tree* as defined in Definition 5.4. The size of the tree is determined by a threshold value, T, associated with each leaf node. This is the maximum diameter allowed for any leaf. Here *diameter* is the average of the pairwise distance between all points in the cluster. Each internal node corresponds to a cluster that is composed of the subclusters represented by its children.

DEFINITION 5.3. A **clustering feature (CF)** is a triple $(N, \vec{LS}, SS)$, where the number of the points in the cluster is N, $\vec{LS}$ is the sum of the points in the cluster, and SS is the sum of the squares of the points in the cluster.

DEFINITION 5.4. A **CF tree** is a balanced tree with a branching factor (maximum number of children a node may have) B. Each internal node contains a CF triple for each of its children. Each leaf node also represents a cluster and contains a CF entry for each subcluster in it. A subcluster in a leaf node must have a diameter no greater than a given threshold value T.

Unlike a dendrogram, a CF tree is searched in a top-down fashion. Each node in the CF tree contains clustering feature information about its subclusters. As points are added to the clustering problem, the CF tree is built. A point is inserted into the cluster (represented by a leaf node) to which it is closest. If the diameter for the leaf node is greater than T, then a splitting and balancing of the tree is performed (similar to that used in a B-tree). The algorithm adapts to main memory size by changing the threshold value. A larger threshold, T, yields a smaller CF tree. This process can be performed without rereading the data. The clustering feature data provides enough information to perform this condensation. The complexity of the algorithm is $O(n)$.

ALGORITHM 5.10

```
Input:
   D = {t1, t2, ..., tn}   //Set of elements
   T      // Threshold for CF tree construction
Output:
   K      //Set of clusters
BIRCH clustering algorithm:
   for each ti ∈ D do
      determine correct leaf node for ti insertion;
      if threshold condition is not violated, then
         add ti to cluster and update CF triples;
      else
         if room to insert ti, then
            insert ti as single cluster and update CF triples;
         else
            split leaf node and redistribute CF features;
```

Algorithm 5.10 outlines the steps performed in BIRCH. Not shown in this algorithm are the parameters needed for the CF tree construction, such as its branching factor, the page block size, and memory size. Based on size, each node has room for a fixed number, B, of clusters (i.e., CF triples). The first step creates the CF tree in memory. The threshold value can be modified if necessary to ensure that the tree fits into the available memory space. Insertion into the CF tree requires scanning the tree from the root down, choosing the node closest to the new point at each level. The distance here is calculated by looking

at the distance between the new point and the centroid of the cluster. This can be easily calculated with most distance measures (e.g., Euclidean or Manhattan) using the CF triple. When the new item is inserted, the CF triple is appropriately updated, as is each triple on the path from the root down to the leaf. It is then added to the closest leaf node found by adjusting the CF value for that node. When an item is inserted into a cluster at the leaf node of the tree, the cluster must satisfy the threshold value. If it does, then the CF entry for that cluster is modified. If it does not, then that item is added to that node as a single-item cluster.

Node splits occur if no space exists in a given node. This is based on the size of the physical page because each node size is determined by the page size. An attractive feature of the CF values is that they are additive; that is, if two clusters are merged, the resulting CF is the addition of the CF values for the starting clusters. Once the tree is built, the leaf nodes of the CF tree represent the current clusters.

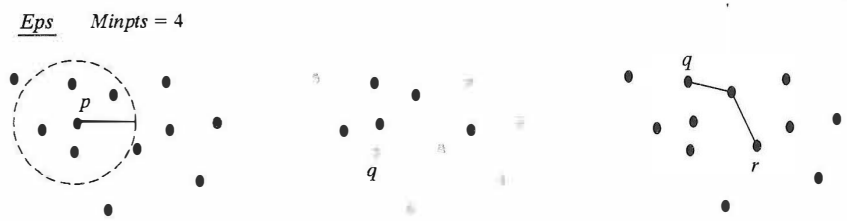
In reality, this algorithm, Algorithm 5.10, is only the first of several steps proposed for the use of BIRCH with large databases. The complete outline of steps is:

1. Create initial CF tree using a modified version of Algorithm 5.10. This in effect "loads" the database into memory. If there is insufficient memory to construct the CF tree with a given threshold, the threshold value is increased and a new smaller CF tree is constructed. This can be done by inserting the leaf nodes of the previous tree into the new small tree.
2. The clustering represented by the CF tree may not be natural because each entry has a limited size. In addition, the input order can negatively impact the results. These problems can be overcome by another global clustering approach applied to the leaf nodes in the CF tree. Here each leaf node is treated as a single point for clustering. Although the original work proposes a centroid-based agglomerative hierarchical clustering algorithm to cluster the subclusters, other clustering algorithms could be used.
3. The last phase (which is optional) reclusters all points by placing them in the cluster that has the closest centroid. Outliers, points that are too far from any centroid, can be removed during this phase.

BIRCH is linear in both space and I/O time. The choice of threshold value is imperative to an efficient execution of the algorithm. Otherwise, the tree may have to be rebuilt many times to ensure that it can be memory-resident. This gives the worst-case time complexity of $O(n^2)$.

5.6.2 DBSCAN

The approach used by *DBSCAN (density-based spatial clustering of applications with noise)* is to create clusters with a minimum size and density. Density is defined as a minimum number of points within a certain distance of each other. This handles the outlier problem by ensuring that an outlier (or a small set of outliers) will not create a cluster. One input parameter, *MinPts*, indicates the minimum number of points in any cluster. In addition, for each point in a cluster there must be another point in the cluster whose distance from it is less than a threshold input value, *Eps*. The *Eps-neighborhood* or *neighborhood* of a point is the set of points within a distance of *Eps*. The desired number of clusters, k, is not input but rather is determined by the algorithm itself.



(a) Eps-neighborhood (b) Core points (c) Density reachable

FIGURE 5.12: DBSCAN example.

DBSCAN uses a new concept of density. We first must look at some definitions from [EKSX96]. Definition 5.5 defines *directly density-reachable*. The first part of the definition ensures that the second point is "close enough" to the first point. The second portion of the definition ensures that there are enough *core points* close enough to each other. These core points form the main portion of a cluster in that they are all close to each other. A directly density-reachable point must be close to one of these core points, but it need not be a core point itself. In that case, it is called a *border point*. A point is said to be *density-reachable* from another point if there is a chain from one to the other that contains only points that are directly density-reachable from the previous point. This guarantees that any cluster will have a core set of points very close to a large number of other points (core points) and then some other points (border points) that are sufficiently close to at least one core point.

DEFINITION 5.5. Given values *Eps* and *MinPts*, a point p is **directly density-reachable** from q if

- $\text{dis}(p, q) \leq Eps$

  and

- $|\{r \mid \text{dis}(r, q) \leq Eps\}| \geq \text{MinPts}$

Figure 5.12 illustrates the concepts used by DBSCAN. This figure shows 12 points. The assumed *Eps* value is illustrated by the straight line. In part (a) it is shown that there are 4 points within the neighborhood of point p. As is seen, p is a core point because it has 4 (*MinPts* value) points within its neighborhood. Part (b) shows the 5 core points in the figure. Note that of the 4 points that are in the neighborhood of p, only 3 are themselves core points. These 4 points are said to be directly density-reachable from p. Point q is not a core point and is thus called a border point. We have partitioned the points into a core set of points that are all close to each other; then border points, which are close to at least one of the core points; and finally the remaining points, which are not close to any core point. Part (C) shows that even though point r is not a core point, it is density-reachable from q.

Algorithm 5.11 outlines the DBSCAN algorithm. Because of the restrictions on what constitutes a cluster when the algorithm finishes, there will be points not assigned to a cluster. These are defined as noise.

ALGORITHM 5.11

```
Input:
   D = {t1, t2, ..., tn}    //Set of elements
   MinPts      // Number of points in cluster
   Eps    // Maximum distance for density measure
Output:
   K = {K1, K2, ..., Kk}      //Set of clusters
DBSCAN algorithm:
   k = 0;    // Initially there are no clusters.
   for i = 1 to n do
      if ti is not in a cluster, then
         X = {tj | tj is density-reachable from ti};
         if X is a valid cluster, then
            k = k + 1;
            Kk = X;
```

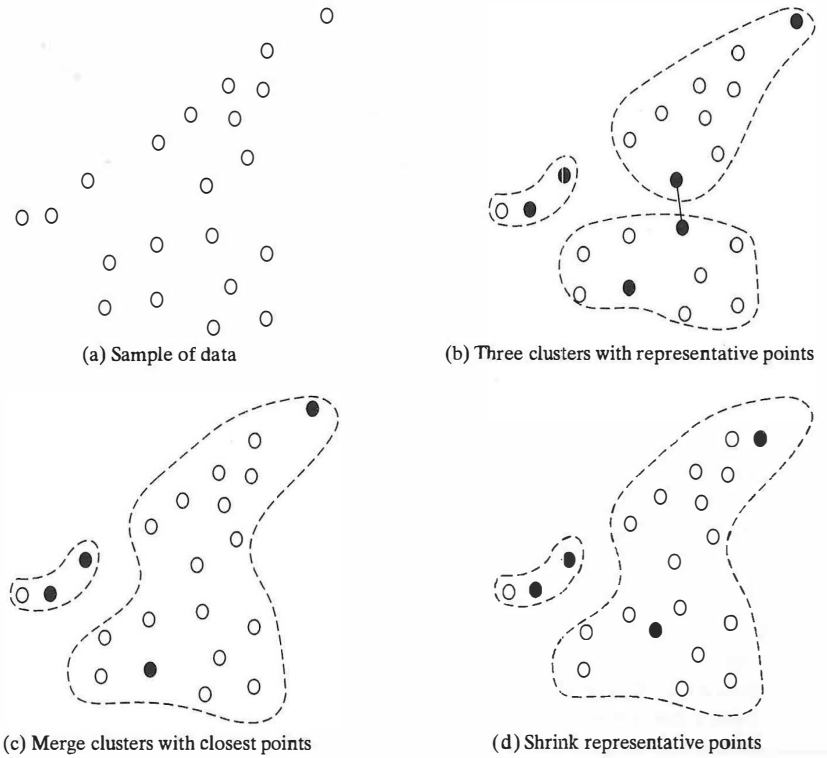
The expected time complexity of DBSCAN is $O(n \lg n)$. It is possible that a border point could belong to two clusters. The stated algorithm will place this point in whichever cluster is generated first. DBSCAN was compared with CLARANS and found to be more efficient by a factor of 250 to 1900 [EKSX96]. In addition, it successfully found all clusters and noise from the test dataset, whereas CLARANS did not.

5.6.3 CURE Algorithm

One objective for the *CURE (Clustering Using REpresentatives)* clustering algorithm is to handle outliers well. It has both a hierarchical component and a partitioning component. First, a constant number of points, c, are chosen from each cluster. These well-scattered points are then shrunk toward the cluster's centroid by applying a shrinkage factor, α. When α is 1, all points are shrunk to just one—the centroid. These points represent the cluster better than a single point (such as a medoid or centroid) could. With multiple representative points, clusters of unusual shapes (not just a sphere) can be better represented. CURE then uses a hierarchical clustering algorithm. At each step in the agglomerative algorithm, clusters with the closest pair of representative points are chosen to be merged. The distance between them is defined as the minimum distance between any pair of points in the representative sets from the two clusters.

The basic approach used by CURE is shown in Figure 5.13. The first step shows a sample of the data. A set of clusters with its representative points exists at each step in the processing. In Figure 5.13(b) there are three clusters, each with two representative points. The representative points are shown as darkened circles. As discussed in the following paragraphs, these representative points are chosen to be far from each other as well as from the mean of the cluster. In part (c), two of the clusters are merged and two new representative points are chosen. Finally, in part (d), these points are shrunk toward the mean of the cluster. Notice that if one representative centroid had been chosen for the clusters, the smaller cluster would have been merged with the bottom cluster instead of with the top cluster.

CURE handles limited main memory by obtaining a random sample to find the initial clusters. The random sample is partitioned, and each partition is then partially clustered. These resulting clusters are then completely clustered in a second pass. The



(a) Sample of data

(b) Three clusters with representative points

(c) Merge clusters with closest points

(d) Shrink representative points

FIGURE 5.13: CURE example.

sampling and partitioning are done solely to ensure that the data (regardless of database size) can fit into available main memory. When the clustering of the sample is complete, the labeling of data on disk is performed. A data item is assigned to the cluster with the closest representative points. The basic steps of CURE for large databases are:

1. Obtain a sample of the database.
2. Partition the sample into p partitions of size $\frac{n}{p}$. This is done to speed up the algorithm because clustering is first performed on each partition.
3. Partially cluster the points in each partition using the hierarchical algorithm (see Algorithm 5.12). This provides a first guess at what the clusters should be. The number of clusters is $\frac{n}{pq}$ for some constant q.
4. Remove outliers. Outliers are eliminated by the use of two different techniques. The first technique eliminates clusters that grow very slowly. When the number of clusters is below a threshold, those clusters with only one or two items are deleted. It is possible that close outliers are part of the sample and would not be identified by the first outlier elimination technique. The second technique removes very small clusters toward the end of the clustering phase.

5. Completely cluster all data in the sample using Algorithm 5.12. Here, to ensure processing in main memory, the input includes only the cluster representatives from the clusters found for each partition during the partial clustering step (3).
6. Cluster the entire database on disk using c points to represent each cluster. An item in the database is placed in the cluster that has the closest representative point to it. These sets of representative points are small enough to fit into main memory, so each of the n points must be compared to ck representative points.

The time complexity of CURE is $O(n^2 \lg n)$, while space is $O(n)$. This is worst-case behavior. The improvements proposed for main memory processing certainly improve on this time complexity because the entire clustering algorithm is performed against only the sample. When clustering is performed on the complete database, a time complexity of only $O(n)$ is required. A heap and k-D tree data structure are used to ensure this performance. One entry in the heap exists for each cluster. Each cluster has not only its representative points, but also the cluster that is closest to it. Entries in the heap are stored in increasing order of the distances between clusters. We assume that each entry u in the heap contains the set of representative points, *u.rep*; the mean of the points in the cluster, *u.mean*; and the cluster closest to it, *u.closest*. We use the heap operations: *heapify* to create the heap, *min* to extract the minimum entry in the heap, *insert* to add a new entry, and *delete* to delete an entry. A *merge* procedure is used to merge two clusters. It determines the new representative points for the new cluster. The basic idea of this process is to first find the point that is farthest from the mean. Subsequent points are then chosen based on being the farthest from those points that were previously chosen. A predefined number of points is picked. A *k-D tree* is a balanced binary tree that can be thought of as a generalization of a binary search tree. It is used to index data of k dimensions where the i^{th} level of the tree indexes the i^{th} dimension. In CURE, a k-D tree is used to assist in the merging of clusters. It stores the representative points for each cluster. Initially, there is only one representative point for each cluster, the sole item in it. Operations performed on the tree are: *delete* to delete an entry form the tree, *insert* to insert an entry into it, and *build* to initially create it. The hierarchical clustering algorithm itself, which is from [GRS98], is shown in Algorithm 5.12. We do not include here either the sampling algorithms or the merging algorithm.

ALGORITHM 5.12

```
Input:
   D = {t1, t2, ..., tn}   //Set of elements
   k      // Desired number of clusters
Output:
   Q      //Heap containing one entry for each cluster
CURE algorithm:
   T = build(D);
   Q = heapify(D); // Initially build heap with one entry per item;
   repeat
      u = min(Q);
      delete(Q, u.close);
      w = merge(u, v);
      delete(T, u);
      delete(T, v);
```



```
      insert(T, w);
      for each x ∈ Q do
         x.close = find closest cluster to x;
         if x is closest to w, then
            w.close = x;
      insert(Q, w);
   until number of nodes in Q is k;
```

Performance experiments compared CURE to BIRCH and the MST approach [GRS98]. The quality of the clusters found by CURE is better. While the value of the shrinking factor α does impact results, with a value between 0.2 and 0.7, the correct clusters are still found. When the number of representative points per cluster is greater than five, the correct clusters are still always found. A random sample size of about 2.5% and the number of partitions is greater than one or two times k seem to work well. The results with large datasets indicate that CURE scales well and outperforms BIRCH.

5.7 CLUSTERING WITH CATEGORICAL ATTRIBUTES

Traditional algorithms do not always work with categorical data. Example 5.8 illustrates some problems that exist when clustering categorical data. This example uses a hierarchical-based centroid algorithm to illustrate the problems. The problem illustrated here is that the centroid tends to weaken the relationship between the associated cluster and other clusters. The problems worsens as more and more clusters are merged. The number of attributes appearing in the mean increases, while the individual values actually decreases. This makes the centroid representations very similar and makes distinguishing between clusters difficult.

EXAMPLE 5.8

Consider an information retrieval system where documents may contain keywords {book, water, sun, sand, swim, read}. Suppose there are four documents, where the first contains the word {book}, the second contains {water, sun, sand, swim}, the third contains {water, sun, swim, read}, and the fourth contains {read, sand}. We can represent the four books using the following boolean points: (1, 0, 0, 0, 0, 0), (0, 1, 1, 1, 1, 0), (0, 1, 1, 0, 1, 1), (0, 0, 0, 1, 0, 1). We can use the Euclidean distance to develop the following adjacency matrix of distances:

	1	2	3	4
1	0	2.24	2.24	1.73
2	2.24	0	1.41	2
3	2.24	1.41	0	2
4	1.73	2	2	0

The distance between points 2 and 3 is the smallest (1.41), and thus they are merged. When they are merged, we get a cluster containing {(0, 1, 1, 1, 1, 0), (0, 1, 1, 0, 1, 1)} with a centroid of (0, 1, 1, 0.5, 1, 0.5). At this point we have a distance from this new cluster centroid to the original points 1 and 4 being 2.24 and 1.73, respectively, while the distance between original points 1 and 4 is 1.73. Thus, we could next merge these points

even though they have no keywords in common. So with $k = 2$ we have the following clusters: $\{\{1, 4\}, \{2, 3\}\}$.

The *ROCK (RObust Clustering using linKs)* clustering algorithm is targeted to both boolean data and categorical data. A novel approach to identifying similarity is based on the number of links between items. A pair of items are said to be *neighbors* if their similarity exceeds some threshold. This need not be defined based on a precise metric, but rather a more intuitive approach using domain experts could be used. The number of *links* between two items is defined as the number of common neighbors they have. The objective of the clustering algorithm is to group together points that have more links. The algorithm is a hierarchical agglomerative algorithm using the number of links as the similarity measure rather than a measure based on distance.

Instead of using a Euclidean distance, a different distance, such as the Jaccard coefficient, has been proposed. One proposed similarity measure based on the Jaccard coefficient is defined as

$$\text{sim}(t_i, t_j) = \frac{|\, t_i \cap t_j \,|}{|\, t_i \cup t_j \,|} \tag{5.16}$$

If the tuples are viewed to be sets of items purchased (i.e., market basket data), then we look at the number of items they have in common divided by the total number in both. The denominator is used to normalize the value to be between 0 and 1.

The number of links between a pair of points can be viewed as the number of unique paths of length 2 between them. The authors argue that the use of links rather than similarity (distance) measures provides a more global approach because the similarity between points is impacted by other points as well. Example 5.9 illustrates the use of links by the ROCK algorithm using the data from Example 5.8 using the Jaccard coefficient. Note that different threshold values for neighbors could be used to get different results. Also note that a hierarchical approach could be used with different threshold values for each level in the dendrogram.

EXAMPLE 5.9

Using the data from Example 5.8, we have the following table of similarities (as opposed to the distances given in the example):

	1	2	3	4
1	1	0	0	0
2	0	1	0.6	0.2
3	0	0.6	1	0.2
4	0	0.2	0.2	1

Suppose we say that the threshold for a neighbor is 0.6, then we have the following are the neighbors: {(2, 3), (2, 4), (3, 4)}. Note that in the following we add to these that a point is a neighbor of itself so that we have the additional neighbors: {(1, 1), (2, 2), (3, 3), (4, 4)}. The following table shows the number of links (common neighbors between points) assuming that the threshold for a neighbor is 0.6:

	1	2	3	4
1	1	0	0	0
2	0	3	3	3
3	0	3	3	3
4	0	3	3	3

In this case, then, we have the following clusters: {{1}, {2, 3, 4}}. Comparing this to the set of clustering found with a traditional Euclidean distance, we see that a "better" set of clusters has been created.

The ROCK algorithm is divided into three general parts:

1. Obtaining a random sample of the data.
2. Performing clustering on the data using the link agglomerative approach. A goodness measure is used to determine which pair of points is merged at each step.
3. Using these clusters the remaining data on disk are assigned to them.

The goodness measure used to merge clusters is:

$$g(K_i, K_j) = \frac{\text{link}(K_i, K_j)}{(n_i + n_j)^{1+2f(\Theta)} - n_i^{1+2f(\Theta)} - n_j^{1+2f(\Theta)}}. \tag{5.17}$$

Here $\text{link}(K_i, K_j)$ is the number of links between the two clusters. Also, n_i and n_j are the number of points in each cluster. The denominator is used to normalize the number of links because larger clusters would be expected to have more links simply because they are larger. $n_i^{1+2f(\Theta)}$ is an estimate for the number of links between pairs of points in K_i when the threshold used for the similarity measure is Θ. The function $f(\Theta)$ depends on the data, but it is found to satisfy the property that each item in K_i has approximately $n_i^{f(\Theta)}$ neighbors in the cluster. Obviously, if all points in the cluster are connected, $f(\Theta) = 1$. Then n_i^3 is the number of links between points in K_i.

The first step in the algorithm converts the adjacency matrix into a boolean matrix where an entry is 1 if the two corresponding points are neighbors. As the adjacency matrix is of size n^2, this is an $O(n^2)$ step. The next step converts this into a matrix indicating the links. This can be found by calculating $S \times S$, which can be done in $O(n^{2.37})$ [GRS99]. The hierarchical clustering portion of the algorithm then starts by placing each point in the sample in a separate cluster. It then successively merges clusters until k clusters are found. To facilitate this processing, both local and global heaps are used. A local heap, q, is created to represent each cluster. Here q contains every cluster that has a nonzero link to the cluster that corresponds to this cluster. Initially, a cluster is created for each point, t_i. The heap for t_i, $q[t_i]$, contains every cluster that has a nonzero link to $\{t_i\}$ The global heap contains information about each cluster. All information in the heap is ordered based on the goodness measure, which is shown in Equation 5.17.

5.8 COMPARISON

The different clustering algorithms discussed in this chapter are compared in Table 5.3. Here we include a classification of the type of algorithm, space and time complexity, and general notes concerning applicability.

TABLE 5.3: Comparison of Clustering Algorithms

Algorithm	Type	Space	Time	Notes
Single link	Hierarchical	$O(n^2)$	$O(kn^2)$	Not incremental
Average link	Hierarchical	$O(n^2)$	$O(kn^2)$	Not incremental
Complete link	Hierarchical	$O(n^2)$	$O(kn^2)$	Not incremental
MST	Hierarchical/ partitional	$O(n^2)$	$O(n^2)$	Not incremental
Squared error	Partitional	$O(n)$	$O(tkn)$	Iterative
K-means	Partitional	$O(n)$	$O(tkn)$	Iterative; No categorical
Nearest neighbor	Partitional	$O(n^2)$	$O(n^2)$	Iterative
PAM	Partitional	$O(n^2)$	$O(tk(n-k)^2)$	Iterative; Adapted agglomerative; Outliers
BIRCH	Partitional	$O(n)$	$O(n)$ (no rebuild)	CF-tree; Incremental; Outliers
CURE	Mixed	$O(n)$	$O(n)$	Heap; k-D tree; Incremental; Outliers; Sampling
ROCK	Agglomerative	$O(n^2)$	$O(n^2 \, lgn)$	Sampling; Categorical; Links
DBSCAN	Mixed	$O(n^2)$	$O(n^2)$	Sampling; Outliers

The single link, complete link, and average link techniques are all hierarchical techniques with $O(n^2)$ time and space complexity. While we discussed the agglomerative versions of these are also divisive versions, which create the clusters in a top-down manner. They all assume the data are present and thus are not incremental. There are several clustering algorithms based on the construction of an MST. There are both hierarchical and partitional versions. Their complexity is identical to that for the other hierarchical techniques, and since they depend on the construction of the MST, they are not incremental. Both K-means and the squared error techniques are iterative, requiring $O(tkn)$ time. The nearest neighbor is not iterative, but the number of clusters is not predetermined. Thus, the worst-case complexity can be $O(n^2)$. BIRCH appears to be quite efficient, but remember that the CF-tree may need to be rebuilt. The time complexity in the table assumes that the tree is not rebuilt. CURE is an improvement on these by using sampling and partitioning to handle scalability well and uses multiple points rather than just one point to represent each cluster. Using multiple points allows the approach to detect nonspherical clusters. With sampling, CURE obtains an $O(n)$ time complexity. However, CURE does not handle categorical data well. This also allows it to be more resistant to the negative impact of outliers. K-means and PAM work by iteratively reassigning items to clusters, which may not find a global optimal assignment. The results of the K-means algorithm is quite sensitive to the presence of outliers. Through the use of the CF-tree, Birch is both dynamic and scalable. However, it detects only spherical type clusters. DBSCAN is a density-based approach. The time complexity of DBSCAN can be improved to $O(n \, lgn)$ with appropriate spatial indices. We have not included the genetic algorithms in this table because their performance totally depends on the technique chosen to represent individuals, how crossover is done, and the termination condition used.

5.9 EXERCISES

1. A major problem with the single link algorithm is that clusters consisting of long chains may be created. Describe and illustrate this concept.
2. Show the dendrogram created by the single, complete, and average link clustering algorithms using the following adjacency matrix:

Item	A	B	C	D
A	0	1	4	5
B	1	0	2	6
C	4	2	0	3
D	5	6	3	0

3. Construct a graph showing all edges for the data in Exercise 2. Find an MST for this graph. Is the MST single link hierarchical clustering the same as that found using the traditional single link algorithm?
4. Convert Algorithm 5.1 to a generic divisive algorithm. What technique would be used to split clusters in the single link and complete link versions?
5. Trace the results of applying the squared error Algorithm 5.5 to the data from Example 5.4 into two clusters. Indicate the convergence threshold you have used.
6. Use the K-means algorithm to cluster the data in Example 5.4 into three clusters.
7. Trace the use of the nearest neighbor algorithm on the data of Exercise 2 assuming a threshold of 3.
8. Determine the cost C_{jih} for each of the four cases given for the PAM algorithm.
9. Finish the application of PAM in Example 5.6.
10. (***Research***) Perform a survey of recently proposed clustering algorithms. Identify where they fit in the classification tree in Figure 5.2. Describe their approach and performance.

5.10 BIBLIOGRAPHIC NOTES

There are many excellent books examining the concept of clustering. In [JD88], a thorough treatment of clustering algorithms, including application domains, and statement of algorithms, is provided. This work also looks at a different classification of clustering techniques. Other clustering and prediction books include [Har75], [JS71], [SS73], [TB70], and [WI98].

A survey article of clustering with a complete list of references was published in 1999 [JMF99]. It covers more clustering techniques than are found in this chapter. Included are fuzzy clustering, evolutionary techniques, and a comparison of the two. An excellent discussion of applications is also included. Fuzzy clustering associates a membership function with every item and every cluster. Imagine that each cluster is

represented as a vector of membership values, one for each element. Other clustering surveys have been published in [NH94] and [HKT01].

Clustering tutorials have been presented at SIGMOD 1999 [HK99] and PAKDD-02.[1]

The agglomerative clustering methods are among the oldest techniques. Proposals include SLINK [Sib73] for single linkage and CLINK [Def77] for complete linkage. An excellent study of these algorithms can be found in [KR90]. The AGNES and DIANA techniques are some of the earliest methods. *AGNES (AGglomerative NESting)* is agglomerative, while *DIANA (DIvisia ANAlysis)* is divisive. Both are known not to scale well. Articles on single link clustering date back to 1951 [FLP+51]. The EM algorithm has frequently been used to perform interative clustering [DLR77]. There have been many variations of the K-means clustering algorithm. The earliest reference is to a version by Forgy in 1965 [For65], [McQ67]. Another approach for partitional clustering is to allow splitting and merging of clusters. Here merging is performed based on the distance between the centroids of two clusters. A cluster is split if its variance is above a certain threshold; One proposed algorithm performing this is called ISODATA [BH65]. CURE, predominantly a hierarchical technique, was first proposed in [GRS98]. Finding connected components in a graph is a well-known graph technique that is described in any graph theory or data structures text such as [Har72].

The MST partitional algorithm was originally proposed by Zahn [Zha71].

A recent work has looked at extending the definition of clustering to be more applicable to categorical data. *CAtegorical ClusTering Using Summaries (CACTUS)* generalizes the traditional definition of clustering and distance. To perform the clustering, it uses summary information obtained from the actual data. The summary information fits into main memory and can be easily constructed. The definition of similarity between tuples is given by looking at the support of two attribute values within the database D. Given two categorical attribute A_i, A_j with domains D_i, D_j, respectively, the *support* of the attribute pair (a_i, a_j) is defined as:

$$\sigma_D(a_i, a_j) = \mid \{t \in D : t.A_i = a_i \wedge t.A_j = a_j\} \mid \tag{5.18}$$

The similarity of attributes used for clustering is based on the support [GGR99b].

Several techniques have been proposed to scale up clustering algorithms to work effectively on large databases. Sampling and data compressions are perhaps the most common techniques. With sampling, the algorithm is applied to a large enough sample to ideally produce a good clustering result. Both BIRCH and CACTUS employ compression techniques. A more recent data compression approach, data bubbles, has been applied to hierarchical clustering [BKKS01]. A *data bubble* is a compressed data item that represents a larger set of items in the database. Given a set of items, a data bubble consists of (a representative item, cardinality of set, radius of set, estimated average k nearest neighbor distances in the set). Another compression technique for hierarchical clustering, which actually compresses the dendrogram, has recently been proposed [XD01b].

A recent article has proposed outlier detection algorithms that scale well to large datasets and can be used to model the previous statistical tests [KN98].

[1] Osmar Zaiane and Andrew Foss, "Data Clustering Analysis, from Simple Groupings to Scalable Clustering with Constraints," Tutorial at Sixth Pacific-Asia Conference on Knowledge Discovery and Data Mining, May 6, 2002.

Discussion of the bond energy algorithm and its use can be found in [WTMSW72], [ÖV99], and [TF82]. It can be used to cluster attributes based on use and then perform logical or physical clustering.

DBSCAN was first proposed in [EKSX96]. Other density-based algorithms include DENCLUE [HK98] and OPTICS [].

ROCK was proposed by Guha in [GRS99].

There have been many approaches targeted to clustering of large databases, including BIRCH [ZRL96], CLARANS [NH94], CURE [GRS98], DBSCAN [EKSX96], and ROCK [GRS99]. Specifics concerning CF tree maintenance can be found in the literature [ZRL96]. A discussion of the use of R*-trees in DBSCAN can be found in [EKSX96]. It is possible that a border point could belong to two clusters. A recent algorithm, *CHAMELEON*, is a hierarchical clustering algorithm that bases merging of clusters on both closeness and their interconnection [KHK99]. When dealing with large databases, the requirement to fix the number of clusters dynamically can be a major problem. Recent research into *online* (or *iterative*) clustering algorithms has been performed. "Online" implies that the algorithm can be performed dynamically as the data are generated, and thus it works well for dynamic databases. In addition, some work has examined how to adapt, thus allowing the user to change the number of clusters dynamically. These *adaptive* algorithms avoid having to completely recluster the database if the users' needs change. One recent online approach represents the clusters by profiles (such as cluster mean and size). These profiles are shown to the user, and the user has the ability to change the parameters (number of clusters) at any time during processing. One recent clustering approach is both online and adaptive, OAK (online adaptive clustering) [XD01b]. OAK can also handle outliers effectively by adjusting a viewing parameter, which gives the user a broader view of the clustering, so that he or she can choose his or her desired clusters.

NNs have been used to solve the clustering problem [JMF99]. Kohonen's self-organizing maps are introduced in [Koh82]. One of the earliest algorithms was the leader clustering algorithm proposed by Hartigan [Har75]. Its time complexity is only $O(kn)$ and its space is only $O(k)$. A common NN applied is a competitive one such as an SOFM where the learning is nonsupervised [JMF99].

References to clustering algorithms based on genetic algorithms include [JB91] and [BRE91].

CHAPTER 6

Association Rules

6.1 INTRODUCTION
6.2 LARGE ITEMSETS
6.3 BASIC ALGORITHMS
6.4 PARALLEL AND DISTRIBUTED ALGORITHMS
6.5 COMPARING APPROACHES
6.6 INCREMENTAL RULES
6.7 ADVANCED ASSOCIATION RULE TECHNQIUES
6.8 MEASURING THE QUALITY OF RULES
6.9 EXERCISES
6.10 BIBLIOGRAPHIC NOTES

6.1 INTRODUCTION

The purchasing of one product when another product is purchased represents an association rule. Association rules are frequently used by retail stores to assist in marketing, advertising, floor placement, and inventory control. Although they have direct applicability to retail businesses, they have been used for other purposes as well, including predicting faults in telecommunication networks. Association rules are used to show the relationships between data items. These uncovered relationships are not inherent in the data, as with functional dependencies, and they do not represent any sort of causality or correlation. Instead, association rules detect common usage of items. Example 6.1 illustrates this.

EXAMPLE 6.1

A grocery store chain keeps a record of weekly transactions where each transaction represents the items bought during one cash register transaction. The executives of the chain receive a summarized report of the transactions indicating what types of items have sold at what quantity. In addition, they periodically request information about what items are commonly purchased together. They find that 100% of the time that PeanutButter is purchased, so is Bread. In addition, 33.3% of the time PeanutButter is purchased, Jelly is also purchased. However, PeanutButter exists in only about 50% of the overall transactions.

A database in which an association rule is to be found is viewed as a set of tuples, where each tuple contains a set of items. For example, a tuple could be {PeanutButter, Bread, Jelly}, which consists of the three items: peanut butter, bread, and

jelly. Keeping grocery story cash register transactions in mind, each item represents an item purchased, while each tuple is the list of items purchased at one time. In the simplest cases, we are not interested in quantity or cost, so these may be removed from the records before processing. Table 6.1 is used throughout this chapter to illustrate different algorithms. Here there are five transactions and five items: {Beer, Bread, Jelly, Milk, PeanutButter}. Throughout this chapter we list items in alphabetical order within a transaction. Although this is not required, algorithms often assume that this sorting is done in a preprocessing step.

The *support* of an item (or set of items) is the percentage of transactions in which that item (or items) occurs. Table 6.2 shows the support for all subsets of items from our total set. As seen, there is an exponential growth in the sets of items. In this case we could have 31 sets of items from the original set of five items (ignoring the empty set). This explosive growth in potential sets of items is an issue that most association

TABLE 6.1: Sample Data to Illustrate Association Rules

Transaction	Items
t_1	Bread, Jelly, PeanutButter
t_2	Bread, PeanutButter
t_3	Bread, Milk, PeanutButter
t_4	Beer, Bread
t_5	Beer, Milk

TABLE 6.2: Support of All Sets of Items Found in Table 6.1

Set	Support	Set	Support
Beer	40	Beer, Bread, Milk	0
Bread	80	Beer, Bread, PeanutButter	0
Jelly	20	Beer, Jelly, Milk	0
Milk	40	Beer, Jelly, PeanutButter	0
PeanutButter	60	Beer, Milk, PeanutButter	0
Beer, Bread	20	Bread, Jelly, Milk	0
Beer, Jelly	0	Bread, Jelly, PeanutButter	20
Beer, Milk	20	Bread, Milk, PeanutButter	20
Beer, PeanutButter	0	Jelly, Milk, PeanutButter	0
Bread, Jelly	20	Beer, Bread, Jelly, Milk	0
Bread, Milk	20	Beer, Bread, Jelly, PeanutButter	0
Bread, PeanutButter	60	Beer, Bread, Milk, PeanutButter	0
Jelly, Milk	0	Beer, Jelly, Milk, PeanutButter	0
Jelly, PeanutButter	20	Bread, Jelly, Milk, PeanutButter	0
Milk, PeanutButter	20	Beer, Bread, Jelly, Milk, PeanutButter	0
Beer, Bread, Jelly	0		

rule algorithms must contend with, as the conventional approach to generating association rules is in actuality counting the occurrence of sets of items in the transaction database.

Note that we are dealing with categorical data. Given a target domain, the underlying set of items usually is known, so that an encoding of the transactions could be performed before processing. As we will see, however, association rules can be applied to data domains other than categorical.

DEFINITION 6.1. Given a set of *items* $I = \{I_1, I_2, \ldots, I_m\}$ and a database of transactions $D = \{t_1, t_2, \ldots, t_n\}$ where $t_i = \{I_{i1}, I_{i2}, \ldots, I_{ik}\}$ and $I_{ij} \in I$, an **association rule** is an implication of the form $X \Rightarrow Y$ where $X, Y \subset I$ are sets of items called *itemsets* and $X \cap Y = \emptyset$.

DEFINITION 6.2. The **support (s)** for an association rule $X \Rightarrow Y$ is the percentage of transactions in the database that contain $X \cup Y$.

DEFINITION 6.3. The **confidence or strength** (α) for an association rule $X \Rightarrow Y$ is the ratio of the number of transactions that contain $X \cup Y$ to the number of transactions that contain X.

A formal definition, from [AIS93], is found in Definition 6.1. We generally are not interested in all implications but only those that are important. Here importance usually is measured by two features called *support* and *confidence* as defined in Definitions 6.2 and 6.3. Table 6.3 shows the support and confidence for several association rules, including those from Example 6.1.

The selection of association rules is based on these two values as described in the definition of the association rule problem in Definition 6.4. Confidence measures the strength of the rule, whereas support measures how often it should occur in the database. Typically, large confidence values and a smaller support are used. For example, look at Bread $\Rightarrow$ PeanutButter in Table 6.3. With $\alpha = 75\%$, this indicates that this rule holds 75% of the time that it could. That is, 3/4 times that Bread occurs, so does PeanutButter. This is a stronger rule than Jelly $\Rightarrow$ Milk because there are no times Milk is purchased when Jelly is bought. An advertising executive probably would not want to base an advertising campaign on the fact that when a person buys Jelly he also buys Milk. Lower values for support may be allowed as support indicates the percentage of time the rule occurs throughout the database. For example, with Jelly $\Rightarrow$ PeanutButter, the confidence is 100% but the support is only 20%. It may be the case that this association

TABLE 6.3: Support and Confidence for Some Association Rules

$X \Rightarrow Y$	s	α
Bread $\Rightarrow$ PeanutButter	60%	75%
PeanutButter $\Rightarrow$ Bread	60%	100%
Beer $\Rightarrow$ Bread	20%	50%
PeanutButter $\Rightarrow$ Jelly	20%	33.3%
Jelly $\Rightarrow$ PeanutButter	20%	100%
Jelly $\Rightarrow$ Milk	0%	0%

rule exists only in 20% of the transactions, but when the *antecedent* Jelly occurs, the *consequent* always occurs. Here an advertising strategy targeted to people who purchase Jelly would be appropriate.

The discussion so far has centered around the use of association rules in the *market basket* area. Example 6.2 illustrates a use for association rules in another domain: telecommunications. This example, although quite simplified from the similar real-world problem, illustrates the importance of association rules in other domains and the fact that support need not always be high.

EXAMPLE 6.2

A telephone company must ensure that a high percentage of all phone calls are made within a certain period of time. Since each phone call must be routed through many switches, it is imperative that each switch work correctly. The failure of any switch could result in a call not being completed or being completed in an unacceptably long period of time. In this environment, a potential data mining problem would be to predict a failure of a node. Then, when the node is predicted to fail, measures can be taken by the phone company to route all calls around the node and replace the switch. To this end, the company keeps a history of calls through a switch. Each call history indicates the success or failure of the switch, associated timing, and error indication. The history contains results of the last and prior traffic through the switch. A transaction of the type ⟨success, failure⟩ indicates that the most recent call could not be handled successfully, while the call before that was handled fine. Another transaction ⟨ERR1, failure⟩ indicates that the previous call was handled but an error occurred, ERR1. This error could be something like excessive time. The data mining problem can then be stated as finding association rules of the type $X \Rightarrow$ Failure. If these types of rules occur with a high confidence, we could predict failure and immediately take the node off-line. Even though the support might be low because the X condition does not frequently occur, most often when it occurs, the node fails with the next traffic.

DEFINITION 6.4. Given a set of *items* $I = \{I_1, I_2, \ldots, I_m\}$ and a database of transactions $D = \{t_1, t_2, \ldots, t_n\}$ where $t_i = \{I_{i1}, I_{i2}, \ldots, I_{ik}\}$ and $I_{ij} \in I$, the **association rule problem** is to identify all association rules $X \Rightarrow Y$ with a minimum support and confidence. These values (s, α) are given as input to the problem.

The efficiency of association rule algorithms usually is discussed with respect to the number of scans of the database that are required and the maximum number of itemsets that must be counted.

6.2 LARGE ITEMSETS

The most common approach to finding association rules is to break up the problem into two parts:

1. Find large itemsets as defined in Definition 6.5.
2. Generate rules from frequent itemsets.

An *itemset* is any subset of the set of all items, I.

DEFINITION 6.5. A **large (frequent) itemset** is an itemset whose number of occurrences is above a threshold, s. We use the notation L to indicate the complete set of large itemsets and l to indicate a specific large itemset.

Once the large itemsets have been found, we know that any interesting association rule, $X \Rightarrow Y$, must have $X \cup Y$ in this set of frequent itemsets. Note that the subset of any large itemset is also large. Because of the large number of notations used in association rule algorithms, we summarize them in Table 6.4. When a specific term has a subscript, this indicates the size of the set being considered. For example, l_k is a large itemset of size k. Some algorithms divide the set of transactions into partitions. In this case, we use p to indicate the number of partitions and a superscript to indicate which partition. For example, D^i is the i^{th} partition of D.

Finding large itemsets generally is quite easy but very costly. The naive approach would be to count all itemsets that appear in any transaction. Given a set of items of size m, there are 2^m subsets. Since we are not interested in the empty set, the potential number of large itemsets is then $2^m - 1$. Because of the explosive growth of this number, the challenge of solving the association rule problem is often viewed as how to efficiently determine all large itemsets. (When $m = 5$, there are potentially 31 itemsets. When $m = 30$, this becomes 1073741823.) Most association rule algorithms are based on smart ways to reduce the number of itemsets to be counted. These potentially large itemsets are called *candidates*, and the set of all counted (potentially large) itemsets is the *candidate itemset (c)*. One performance measure used for association rule algorithms is the size of C. Another problem to be solved by association rule algorithms is what data structure is to be used during the counting process. As we will see, several have been proposed. A trie or hash tree are common.

When all large itemsets are found, generating the association rules is straightforward. Algorithm 6.1, which is modified from [AS94], outlines this technique. In this algorithm we use a function *support*, which returns the support for the input itemset.

TABLE 6.4: Association Rule Notation

Term	Description
D	Database of transactions
t_i	Transaction in D
s	Support
α	Confidence
X, Y	Itemsets
$X \Rightarrow Y$	Association rule
L	Set of large itemsets
l	Large itemset in L
C	Set of candidate itemsets
p	Number of partitions

ALGORITHM 6.1

```
Input:
   D      //Database of transactions
   I      //Items
   L      //Large itemsets
   s      //Support
   α      //Confidence
Output:
   R      //Association Rules satisfying s and α
ARGen algorithm:
   R = ∅;
   for each l ∈ L do
      for each x ⊂ l such that x ≠ ∅ do
         if support(l)/support(x) ≥ α then
            R = R ∪ {x ⇒ (l − x)};
```

To illustrate this algorithm, again refer to the data in Table 6.1 with associated supports shown in Table 6.2. Suppose that the input support and confidence are $s = 30\%$ and $\alpha = 50\%$, respectively. Using this value of s, we obtain the following set of large itemsets:

$$L = \{\{\text{Beer}\}, \{\text{Bread}\}, \{\text{Milk}\}, \{\text{PeanutButter}\}\{\text{Bread, PeanutButter}\}\}.$$

We now look at what association rules are generated from the last large itemset. Here $l =$ {Bread, PeanutButter}. There are two nonempty subsets of l: {Bread} and {PeanutButter}. With the first one we see:

$$\frac{\text{support}(\{\text{Bread, PeanutButter}\})}{\text{support}(\{\text{Bread}\})} = \frac{60}{80} = 0.75$$

This means that the confidence of the association rule Bread $\Rightarrow$ PeanutButter is 75%, just as is seen in Table 6.3. Since this is above α, it is a valid association rule and is added to R. Likewise with the second large itemset

$$\frac{\text{support}(\{\text{Bread, PeanutButter}\})}{\text{support}(\{\text{PeanutButter}\})} = \frac{60}{60} = 1$$

This means that the confidence of the association rule PeanutButter $\Rightarrow$ Bread is 100%, and this is a valid association rule.

All of the algorithms discussed in subsequent sections look primarily at ways to efficiently discover large itemsets.

6.3 BASIC ALGORITHMS

6.3.1 Apriori Algorithm

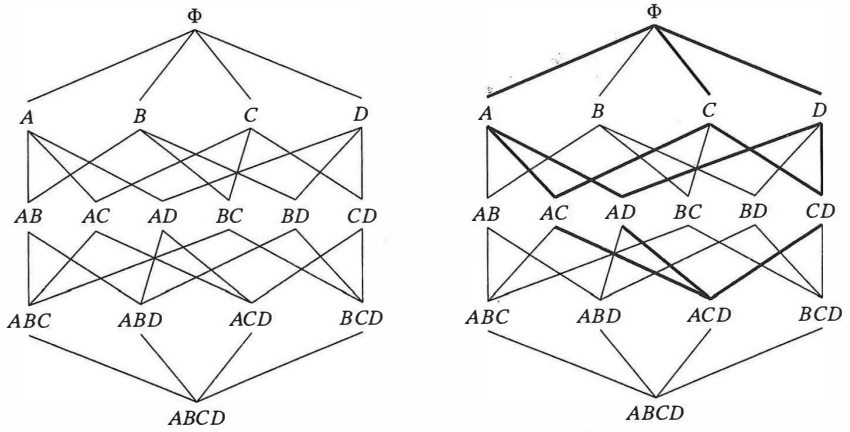
The Apriori algorithm is the most well known association rule algorithm and is used in most commercial products. It uses the following property, which we call the *large itemset property*:

Any subset of a large itemset must be large.

The large itemsets are also said to be *downward closed* because if an itemset satisfies the minimum support requirements, so do all of its subsets. Looking at the contrapositive of this, if we know that an itemset is small, we need not generate any supersets of it as candidates because they also must be small. We use the lattice shown in Figure 6.1(a) to illustrate the concept of this important property. In this case there are four items $\{A, B, C, D\}$. The lines in the lattice represent the subset relationship, so the large itemset property says that any set in a path above an itemset must be large if the original itemset is large. In Figure 6.1 (b) the nonempty subsets of ACD[1] are seen as $\{AC, AD, CD, A, C, D\}$. If ACD is large, so is each of these subsets. If any one of these subsets is small, then so is ACD.

The basic idea of the Apriori algorithm is to generate candidate itemsets of a particular size and then scan the database to count these to see if they are large. During scan i, candidates of size i, C_i are counted. Only those candidates that are large are used to generate candidates for the next pass. That is L_i are used to generate C_{i+1}. An itemset is considered as a candidate only if all its subsets also are large. To generate candidates of size $i+1$, joins are made of large itemsets found in the previous pass. Table 6.5 shows the process using the data found in Table 6.1 with $s = 30\%$ and $\alpha = 50\%$. There are no candidates of size three because there is only one large itemset of size two.

An algorithm called Apriori-Gen is used to generate the candidate itemsets for each pass after the first. All singleton itemsets are used as candidates in the first pass. Here the set of large itemsets of the previous pass, L_{i-1}, is joined with itself to determine the candidates. Individual itemsets must have all but one item in common in order to be combined. Example 6.3 further illustrates the concept. After the first scan, every large itemset is combined with every other large itemset.



(a) Lattice of itemsets for $\{A, B, C, D\}$

(b) Subsets of ACD

FIGURE 6.1: Downward closure.

[1]Following the usual convention with association rule discussions, we simply list the items in the set rather than using the traditional set notation. So here we use ACD to mean $\{A, C, D\}$.

TABLE 6.5: Using Apriori with Transactions in Table 6.1

Pass	Candidates	Large Itemsets
1	{Beer}, {Bread}, {Jelly}, {Milk}, {PeanutButter}	{Beer}, {Bread}, {Milk}, {PeanutButter}
2	{Beer, Bread}, {Beer, Milk}, {Beer, PeanutButter}, {Bread, Milk}, {Bread, PeanutButter}, {Milk, PeanutButter}	{Bread, PeanutButter}

TABLE 6.6: Sample Clothing Transactions

Transaction	Items	Transaction	Items
t_1	Blouse	t_{11}	TShirt
t_2	Shoes, Skirt, TShirt	t_{12}	Blouse, Jeans, Shoes, Skirt, TShirt
t_3	Jeans, TShirt	t_{13}	Jeans, Shoes, Shorts, TShirt
t_4	Jeans, Shoes, TShirt	t_{14}	Shoes, Skirt, TShirt
t_5	Jeans, Shorts	t_{15}	Jeans, TShirt
t_6	Shoes, TShirt	t_{16}	Skirt, TShirt
t_7	Jeans, Skirt	t_{17}	Blouse, Jeans, Skirt
t_8	Jeans, Shoes, Shorts, TShirt	t_{18}	Jeans, Shoes, Shorts, TShirt
t_9	Jeans	t_{19}	Jeans
t_{10}	Jeans, Shoes, TShirt	t_{20}	Jeans, Shoes, Shorts, TShirt

EXAMPLE 6.3

A woman's clothing store has 10 cash register transactions during one day, as shown in Table 6.6. When Apriori is applied to the data, during scan one, we have six candidate itemsets, as seen in Table 6.7. Of these, 5 candidates are large. When Apriori-Gen is applied to these 5 candidates, we combine every one with all the other 5. Thus, we get a total of $4 + 3 + 2 + 1 = 10$ candidates during scan two. Of these, 7 candidates are large. When we apply Apriori-Gen at this level, we join any set with another set that has one item in common. Thus, {Jeans, Shoes} is joined with {Jeans, Shorts} but not with {Shorts, TShirt}. {Jeans, Shoes} will be joined with any other itemset containing either Jeans or Shoes. When it is joined, the new item is added to it. There are four large itemsets after scan four. When we go to join these we must match on two of the three attributes. For example {Jeans, Shoes, Shorts} After scan four, there is only one large itemset. So we obtain no new itemsets of size five to count in the next pass. joins with {Jeans, Shoes, TShirt} to yield new candidate {Jeans, Shoes, Shorts, TShirt}.

The Apriori-Gen algorithm is shown in Algorithm 6.2. Apriori-Gen is guaranteed to generate a superset of the large itemsets of size i, $C_i \supset L_i$, when input L_{i-1}. A

TABLE 6.7: Apriori-Gen Example

Scan	Candidates	Large Itemsets
1	{Blouse}, {Jeans}, {Shoes}, {Shorts}, {Skirt}, {TShirt}	{Jeans}, {Shoes}, {Shorts} {Skirt}, {Tshirt}
2	{Jeans, Shoes}, {Jeans, Shorts}, {Jeans, Skirt}, {Jeans, TShirt}, {Shoes, Shorts}, {Shoes, Skirt}, {Shoes, TShirt}, {Shorts, Skirt}, {Shorts, TShirt}, {Skirt, TShirt}	{Jeans, Shoes}, {Jeans, Shorts}, {Jeans, TShirt}, {Shoes, Shorts}, {Shoes, TShirt}, {Shorts, TShirt}, {Skirt, TShirt}
3	{Jeans, Shoes, Shorts}, {Jeans, Shoes, TShirt}, {Jeans, Shorts, TShirt}, {Jeans, Skirt, TShirt}, {Shoes, Shorts, TShirt}, {Shoes, Skirt, TShirt}, {Shorts, Skirt, TShirt}	{Jeans, Shoes, Shorts}, {Jeans, Shoes, TShirt}, {Jeans, Shorts, TShirt}, {Shoes, Shorts, TShirt}
4	{Jeans, Shoes, Shorts, TShirt}	{Jeans, Shoes, Shorts, TShirt}
5	$\emptyset$	$\emptyset$

pruning step, not shown, could be added at the end of this algorithm to prune away any candidates that have subsets of size $i-1$ that are not large.

ALGORITHM 6.2

```
Input:
   L_{i-1}   //Large itemsets of size i-1
Output:
   C_i       //Candidates of size i
Apriori-gen algorithm:
   C_i = ∅;
   for each I ∈ L_{i-1} do
      for each J ≠ I ∈ L_{i-1} do
         if i-2 of the elements in I and J are equal then
            C_k = C_k ∪ {I ∪ J};
```

Given the large itemset property and Apriori-Gen, the Apriori algorithm itself (see Algorithm 6.3) is rather straightforward. In this algorithm we use c_i to be the count for item $I_i \in I$.

ALGORITHM 6.3

```
Input:
   I       //Itemsets
   D       //Database of transactions
   s       //Support
Output:
   L       //Large itemsets
Apriori algorithm:
   k = 0; //k is used as the scan number.
   L = ∅;
   C_1 = I;        //Initial candidates are set to be the items.
   repeat
      k = k + 1;
      L_k = ∅;
      for each I_i ∈ C_k do
         c_i = 0;   // Initial counts for each itemset are 0.
      for each t_j ∈ D do
         for each I_i ∈ C_K do
            if I_i ∈ t_j then
               c_i = c_i + 1;
      for each I_i ∈ C_k do
         if c_i ≥ (s× | D |) do
            L_k = L_k ∪ I_i;
      L = L ∪ L_k;
      C_{k+1} = Apriori-Gen(L_k)
   until C_{k+1} = ∅;
```


The Apriori algorithm assumes that the database is memory-resident. The maximum number of database scans is one more than the cardinality of the largest large itemset. This potentially large number of database scans is a weakness of the Apriori approach.

6.3.2 Sampling Algorithm

To facilitate efficient counting of itemsets with large databases, sampling of the database may be used. The original sampling algorithm reduces the number of database scans to one in the best case and two in the worst case. The database sample is drawn such that it can be memory-resident. Then any algorithm, such as Apriori, is used to find the large itemsets for the sample. These are viewed as *potentially large* (*PL*) itemsets and used as candidates to be counted using the entire database. Additional candidates are determined by applying the *negative border* function, BD^-, against the large itemsets from the sample. The entire set of candidates is then $C = BD^-(PL) \cup PL$. The negative border function is a generalization of the Apriori-Gen algorithm. It is defined as the minimal set of itemsets that are not in *PL*, but whose subsets are all in *PL*. Example 6.4 illustrates the idea.

EXAMPLE 6.4

Suppose the set of items is $\{A, B, C, D\}$. The set of large itemsets found to exist in a sample of the database is $PL = \{A, C, D, CD\}$. The first scan of the entire database, then, generates the set of candidates as follows: $C = BD^-(PL) \cup PL = \{B, AC, AD\} \cup \{A, C, D, CD\}$. Here we add AC because both A and C are in *PL*. Likewise we add AD. We could not have added ACD because neither AC nor AD is in *PL*. When looking at the lattice (Figure 6.2), we add only sets where all subsets are already in *PL*. Note that we add B because all its subsets are vacuously in *PL*.

Algorithm 6.4 shows the sampling algorithm. Here the Apriori algorithm is shown to find the large itemsets in the sample, but any large itemset algorithm could be used. Any algorithm to obtain a sample of the database could be used as well. The set of large

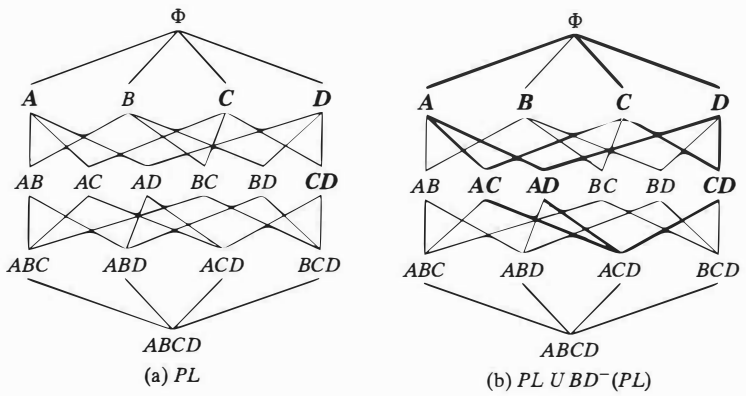


FIGURE 6.2: Negative border.

itemsets is used as a set of candidates during a scan of the entire database. If an itemset is large in the sample, it is viewed to be potentially large in the entire database. Thus, the set of large itemsets from the sample is called PL. In an attempt to obtain all the large itemsets during the first scan, however, PL is expanded by its negative border. So the total set of candidates is viewed to be $C = PL \cup BD^-(PL)$.

During the first scan of the database, all candidates in C are counted. If all candidates that are large are in PL [none in $BD^-(PL)$], then all large itemsets are found. If, however, some large itemsets are in the negative border, a second scan is needed. Think of $BD^-(PL)$ as a buffer area on the border of large itemsets. The set of all itemsets is divided into four areas: those that are known to be large, those that are known to be small, those that are on the negative border of those known to be large, and the others. The negative border is a buffer zone between those known to be large and the others. It represents the smallest possible set of itemsets that could potentially be large. Because of the large itemset property, we know that if there are no large itemsets in this area, then there can be none in the rest of the set.

During the second scan, additional candidates are generated and counted. This is done to ensure that all large itemsets are found. Here ML, the missing large itemsets, are those in L that are not in PL. Since there are some large itemsets in ML, there may be some in the rest of the set of itemsets. To find all the remaining large itemsets in the second scan, the sampling algorithm repeatedly applies the negative border function until the set of possible candidates does not grow further. While this creates a potentially large set of candidates (with many not large), it does guarantee that only one more database scan is required.

ALGORITHM 6.4

```
Input:
   I      //Itemsets
   D      //Database of transactions
   s      //Support
Output:
   L      //Large itemsets
```



```
Sampling algorithm:
   D_S = Sample drawn from D;
   PL = Apriori(I, D_S, smalls);
   C = PL ∪ BD⁻(PL);
   L = ∅;
   for each I_i ∈ C do
      c_i = 0;        // Initial counts for each itemset are 0;
   for each t_j ∈ D do            // First scan count.
      for each I_i ∈ C do
         if I_i ∈ t_j, then
            c_i = c_i + 1;
   for each I_i ∈ C do
      if c_i ≥ (s× | D |) do
         L = L ∪ I_i;
   ML = {x | x ∈ BD⁻(PL) ∧ x ∈ L};              //Missing large itemsets.
   if ML ≠ ∅, then
      C = L;        // Set candidates to be the large itemsets.
      repeat
         C = C ∪ BD⁻(C);          // Expand candidate sets
                                     using negative border.
      until no new itemsets are added to C;
      for each I_i ∈ C do
         c_i = 0;              // Initial counts for each itemset are 0.
      for each t_j ∈ D do              // Second scan count.
         for each I_i ∈ C do
            if I_i ∈ t_j, then
               c_i = c_i + 1;
      if c_i ≥ (s× | D |) do
         L = L ∪ I_i;
```

The algorithm shows that the application of the Apriori algorithm to the sample is performed using a support called *smalls*. Here *smalls* can be any support values less than s. The idea is that by reducing the support when finding large itemsets in the sample, more of the true large itemsets from the complete database will be discovered. We illustrate the use of the sampling algorithm on the grocery store data in Example 6.5.

EXAMPLE 6.5

We use the sampling algorithm to find all large itemsets in the grocery data where $s = 20\%$. Suppose that the sample database is determined to be the first two transactions:

$$D_S = \{t_1 = \{\text{Bread, Jelly, PeanutButter}\}, t_2 = \{\text{Bread, PeanutButter}\}\}$$

If we reduce s to be smalls $= 10\%$, then for an itemset to be large in the sample it must occur in at least 0.1×2 transactions. So it must occur in one of the two transactions. When we apply *Apriori* to D_S we get:

$$PL = \{\{\text{Bread}\}, \{\text{Jelly}\}, \{\text{PeanutButter}\}, \{\text{Bread, Jelly}\}, \{\text{Bread, PeanutButter}\}, \{\text{Jelly, PeanutButter}\}, \{\text{Bread, Jelly, PeanutButter}\}\}$$

When we apply the negative border, we get

$$BD^-(PL) = \{\{\text{Beer}\}, \{\text{Milk}\}\}$$

We thus use the following set of candidates to count during the first database scan:

$$PL = \{\{\text{Bread}\}, \{\text{Jelly}\}, \{\text{PeanutButter}\}, \{\text{Bread, Jelly}\}, \{\text{Bread, PeanutButter}\}, \{\text{Jelly, PeanutButter}\}, \{\text{Bread, Jelly, PeanutButter}\}, \{\text{Beer}\}, \{\text{Milk}\}\}$$

Remember that during this scan we use $s = 20\%$ and apply it against all five transactions in the entire database. For an itemset to be large, then, we must have an itemset in $20\% \times 5$ or at least one transaction. We then find that both {Beer} and {Milk} are large. Thus, $ML = \{\{\text{Beer}\}, \{\text{Milk}\}\}$. Following the algorithm, we first set $C = L$, which in this case is also PL. Applying the negative border, we get

$$C = BD^-(C) = \{\{\text{Beer, Bread}\}, \{\text{Beer, Jelly}\}, \{\text{Beer, Milk}\}, \{\text{Beer, PeanutButter}\}, \{\text{Bread, Milk}\}, \{\text{Jelly, Milk}\}, \{\text{Milk, PeanutButter}\}, \{\text{Jelly, Milk}\}\}$$

Since this has uncovered new itemsets, we again apply it and this time find all itemsets of size three. A last application then finds all itemsets of size four, and we scan the database using all remaining itemsets not already known to be large.

Example 6.5 illustrates a potential problem with the use of the sampling algorithm; that is, a very large set of candidates may be used during the second scan. This is required to ensure that all large itemsets are found during the second scan. However, the set of candidates generated by successive applications of the negative border function will not always generate the entire set of itemsets. This happened in Example 6.5 because we found that all itemsets in PL were large and all itemsets in $BD^-(PL)$ also were large. If instead of using a support of 20%, we had used one of 40%, the results would be different, as shown in Example 6.6.

EXAMPLE 6.6

Suppose that smalls and D_S are as were used in Example 6.5. We thus find that PL and $BD^-(PL)$ are the same. During the first scan of the entire database, we identify large itemsets only if they have a support of at least 40%. Looking at Table 6.2, we see that from the initial scan we obtain the following large itemsets:

$$L = \{\{\text{Bread}\}, \{\text{PeanutButter}\}, \{\text{Bread, PeanutButter}\}, \{\text{Beer}\}, \{\text{Milk}\}\}$$

Here $ML = \{\{\text{Beer}\}, \{\text{Milk}\}\}$, so we need a second scan. When we first apply the negative border we get

$$C = BD^-(C) = \{\{\text{Beer, Bread}\}, \{\text{Beer, Milk}\}, \{\text{Beer, PeanutButter}\}, \{\text{Bread, Milk}\}, \{\text{Milk, PeanutButter}\}\}$$

Note that this is smaller than what we found with Example 6.5. Since Jelly is missing, we will not generate the entire set of itemsets during repeated application of BD^-. The second application yields

$$C = BD^-(C) = \{\{\text{Beer, Bread, Milk}\}, \{\text{Beer, Bread, PeanutButter}\}, \{\text{Beer, Milk, PeanutButter}\}, \{\text{Bread, Milk, PeanutButter}\}\}$$

The final application obtains

$$C = BD^-(C) = \{\{\text{Beer, Bread, Milk, PeanutButter}\}\}$$

6.3.3 Partitioning

Various approaches to generating large itemsets have been proposed based on a partitioning of the set of transactions. In this case, D is divided into p partitions $D^1, D^2, \ldots, D^p$. Partitioning may improve the performance of finding large itemsets in several ways:

- By taking advantage of the large itemset property, we know that a large itemset must be large in at least one of the partitions. This idea can help to design algorithms more efficiently than those based on looking at the entire database.
- Partitioning algorithms may be able to adapt better to limited main memory. Each partition can be created such that it fits into main memory. In addition, it would be expected that the number of itemsets to be counted per partition would be smaller than those needed for the entire database.
- By using partitioning, parallel and/or distributed algorithms can be easily created, where each partition could be handled by a separate machine.
- Incremental generation of association rules may be easier to perform by treating the current state of the database as one partition and treating the new entries as a second partition.

The basic partition algorithm reduces the number of database scans to two and divides the database into partitions such that each can be placed into main memory. When it scans the database, it brings that partition of the database into main memory and counts the items in that partition alone. During the first database scan, the algorithm finds all large itemsets in each partition. Although any algorithm could be used for this purpose, the original proposal assumes that some level-wise approach, such as Apriori, is used. Here L^i represents the large itemsets from partition D^i. During the second scan, only those itemsets that are large in at least one partition are used as candidates and counted to determine if they are large across the entire database. Algorithm 6.5 shows this basic partition algorithm.

D^1	t_1	Bread, Jelly, PeanutButter	L^1 = {{Bread}, {Jelly}, {PeanutButter}, {Bread, Jelly}, {Bread, PeanutButter}, {Jelly, PeanutButter}, {Bread, Jelly, PeanutButter}}
	t_2	Bread, PeanutButter	
D^2	t_3	Bread, Milk, PeanutButter	L^2 = {{Beer}, {Bread}, {Milk}, {PeanutButter}, {Beer, Bread}, {Beer, Milk}, {Bread, Milk}, {Bread, PeanutButter}, {Milk, PeanutButter}, {Bread, Milk, PeanutButter}}
	t_4	Beer, Bread	
	t_5	Beer, Milk	

FIGURE 6.3: Partitioning example.

ALGORITHM 6.5

```
Input:
   I      //Itemsets
   D = {D^1, D^2, ..., D^p}          //Database of transactions divided
                                        into partitions
   s      //Support
Output:
   L      //Large itemsets
Partition algorithm:
   C = ∅;
   for i = 1 to p do  //Find large itemsets in each partition.
      L^i = Apriori(I, D^i, s);
      C = C ∪ L^i;
   L = ∅;
   for each I_i ∈ C do
      c_i = 0;        //Initial counts for each itemset are 0.
   for each t_j ∈ D do //    Count candidates during second scan.
      for each I_i ∈ C do
         if I_i ∈ t_j, then
            c_i = c_i + 1;
   for each I_i ∈ C do
      if c_i ≥ (s× | D |) do
         L = L ∪ I_i;
```

Figure 6.3 illustrates the use of the partition algorithm using the market basket data. Here the database is partitioned into two parts, the first containing two transactions and the second with three transactions. Using a support of 10%, the resulting large itemsets L^1 and L^2 are shown. If the items are uniformly distributed across the partitions, then a large fraction of the itemsets will be large. However, if the data are not uniform, there may be a large percentage of false candidates.

6.4 PARALLEL AND DISTRIBUTED ALGORITHMS

Most parallel or distributed association rule algorithms strive to parallelize either the data, known as *data parallelism*, or the candidates, referred to as *task parallelism*. With task parallelism, the candidates are partitioned and counted separately at each processor. Obviously, the partition algorithm would be easy to parallelize using the task parallelism approach. Other dimensions in differentiating different parallel association rule algorithms are the load-balancing approach used and the architecture. The data parallelism algorithms have reduced communication cost over the task, because only the initial candidates (the set of items) and the local counts at each iteration must be distributed. With task parallelism, not only the candidates but also the local set of transactions must be broadcast to all other sites. However, the data parallelism algorithms require that memory at each processor be

large enough to store all candidates at each scan (otherwise the performance will degrade considerably because I/O is required for both the database and the candidate set). The task parallelism approaches can avoid this because only the subset of the candidates that are assigned to a processor during each scan must fit into memory. Since not all partitions of the candidates must be the same size, the task parallel algorithms can adapt to the amount of memory at each site. The only restriction is that the total size of all candidates be small enough to fit into the total size of memory in all processors combined. Note that there are some variations of the basic algorithms discussed in this section that address these memory issues. Performance studies have shown that the data parallelism tasks scale linearly with the number of processors and the database size. Because of the reduced memory requirements, however, the task parallelism may work where data parallelism may not work.

6.4.1 Data Parallelism

One data parallelism algorithm is the *count distribution algorithm (CDA)*. The database is divided into p partitions, one for each processor. Each processor counts the candidates for its data and then broadcasts its counts to all other processors. Each processor then determines the global counts. These then are used to determine the large itemsets and to generate the candidates for the next scan. The algorithm is shown in Algorithm 6.6.

ALGORITHM 6.6

```
Input:
   I      //Itemsets
   P^1, P^2, ..., P^p;          //Processors
   D = D^1, D^2, ..., D^p;          //Database divided into partitions
   s      //Support
Output:
   L      //Large itemsets
Count distribution algorithm:
   perform in parallel at each processor P^l; //Count in parallel.
      k = 0;        //k is used as the scan number.
      L = ∅;
      C_1 = I;      //Initial candidates are set to be the items.
      repeat
         k = k + 1;
         L_k = ∅;
         for each I_i ∈ C_k do
            c_i^l = 0;          //Initial counts for each itemset are 0.
         for each t_j ∈ D^l do
            for each I_i ∈ C_K do
               if I_i ∈ t_j then
                  c_i^l = c_i^l + 1;
         broadcast c_i^l to all other processors;
         for each I_i ∈ C_k do      //Determine global counts.
            c_i = Σ_{l=1}^{p} c_i^l;
         for each I_i ∈ C_k do
            if c_i ≥ (s× | D^1 ∪ D^2 ∪ ··· ∪ D^p |) do
               L_k = L_k ∪ I_i;
         L = L ∪ L_k;
         C_{k+1} = Apriori-Gen(L_k)
      until C_{k+1} = ∅;
```

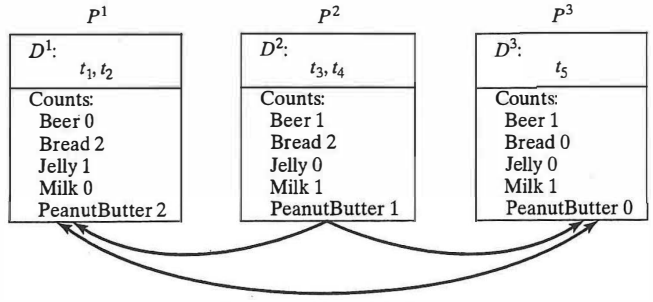


FIGURE 6.4: Data parallelism using CDA (modified from [DXGH00]).

Figure 6.4, which is modified from [DXGH00], illustrates the approach used by the CDA algorithm using the grocery store data. Here there are three processors. The first two transactions are counted at P^1, the next two at P^2, and the last one at P^3. When the local counts are obtained, they are then broadcast to the other sites so that global counts can be generated.

6.4.2 Task Parallelism

The *data distribution algorithm (DDA)* demonstrates task parallelism. Here the candidates as well as the database are partitioned among the processors. Each processor in parallel counts the candidates given to it using its local database partition. Following our convention, we use C_k^l to indicate the candidates of size k examined at processor P^l. Also, L_k^l are the local large k-itemsets at processor l. Then each processor broadcasts its database partition to all other processors. Each processor then uses this to obtain a global count for its data and broadcasts this count to all other processors. Each processor then can determine globally large itemsets and generate the next candidates. These candidates then are divided among the processors for the next scan. Algorithm 6.7 shows this approach. Here we show that the candidates are actually sent to each processor. However, some prearranged technique could be used locally by each processor to determine its own candidates. This algorithm suffers from high message traffic whose impact can be reduced by overlapping communication and processing.

ALGORITHM 6.7

```
Input:
   I      //Itemsets
   P^1, P^2, ..., P^p;          //Processors
   D = D^1, D^2, ..., D^p;          //Database divided into partitions
   s      //Support
Output:
   L      //Large itemsets
Data distribution algorithm:
   C_1 = I;
```



```
for each 1 ≤ l ≤ p do      //Distribute size 1 candidates
                                to each processor.
   determine C_1^l and distribute to P^l;
perform in parallel at each processor P^l //Count in parallel.
   k = 0;       //k is used as the scan number.
   L = ∅;
   repeat
      k = k + 1;
      L_k^l = ∅;
      for each I_i ∈ C_k^l do
         c_i^l = 0;          //Initial counts for each itemset are 0.
      for each t_j ∈ D^l do
         for each I_i ∈ C_k^l do
            if I_i ∈ t_j, then
               c_i^l = c_i^l + 1;          //Determine local counts.
      broadcast D^l to all other processors;
      for every other processor m ≠ l do
         for each t_j ∈ D^m do
            for each I_i ∈ C_k^l do
               if I_i ∈ t_j, then
                     c_i^l = c_i^l + 1;          //Determine
                                                    global counts.
      if c_i ≥ (s× | D^1 ∪ D^2 ∪ ··· ∪ D^p |) do
         L_k^l = L_k^l ∪ I_i;
      broadcast L_k^l to all other processors;
      L_k = L_k^1 ∪ L_k^2 ∪ ··· ∪ L_k^p;          //Global large
                                                    k-itemsets.
      C_{k+1} = Apriori-gen(L_k)
      C_{k+1}^l ⊂ C_{k+1};          //Determine next set of
                                       local candidates.
   until C_{k+1}^l = ∅ ;
```

Figure 6.5, which is modified from [DXGH00], illustrates the approach used by the DDA algorithm using the grocery store data. Here there are three processors. P^1 is counting Beer and Bread, P^2 is counting Jelly and Milk, and P^3 is counting PeanutButter. The first two transactions initially are counted at P^1, the next two at P^2, and the last one at P^3. When the local counts are obtained, the database partitions are then broadcast to the other sites so that each site can obtain a global count.

6.5 COMPARING APPROACHES

Although we have covered only the major association rule algorithms that have been proposed, there have been many such algorithms (see Bibliography). Algorithms can be classified along the following dimensions [DXGH00]:

- **Target:** The algorithms we have examined generate all rules that satisfy a given support and confidence level. Alternatives to these types of algorithms are those that generate some subset of the algorithms based on the constraints given.

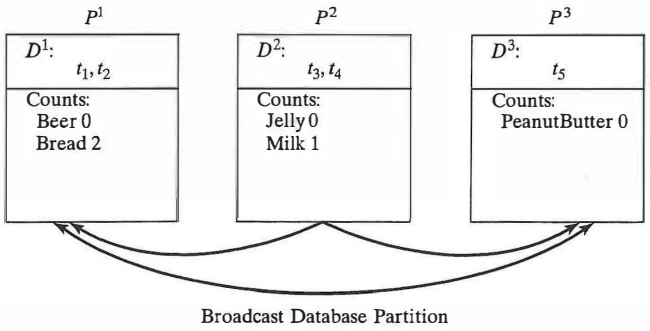


FIGURE 6.5: Task parallelism using DDA (modified from [DXGH00]).

- **Type:** Algorithms may generate regular association rules or more advanced association rules such as those introduced in section 6.7 and Chapters 8 and 9.
- **Data type:** We have examined rules generated for data in categorical databases. Rules may also be derived for other types of data such as plain text. This concept is further investigated in Section 6.7 and in Chapter 7 when we look at Web usage mining.
- **Data source:** Our investigation has been limited to the use of association rules for market basket data. This assumes that data are present in a transaction. The absence of data may also be important.
- **Technique:** The most common strategy to generate association rules is that of finding large itemsets. Other techniques may also be used.
- **Itemset strategy:** Itemsets may be counted in different ways. The most naive approach is to generate all itemsets and count them. As this is usually too space-intensive, the bottom-up approach used by Apriori, which takes advantage of the large itemset property, is the most common approach. A top-down technique could also be used.
- **Transaction strategy:** To count the itemsets, the transactions in the database must be scanned. All transactions could be counted, only a sample may be counted, or the transactions could be divided into partitions.
- **Itemset data structure:** The most common data structure used to store the candidate itemsets and their counts is a hash tree. Hash trees provide an effective technique to store, access, and count itemsets. They are efficient to search, insert, and delete itemsets. A *hash tree* is a multiway search tree where the branch to be taken at each level in the tree is determined by applying a hash function as opposed to comparing key values to branching points in the node. A leaf node in the hash tree contains the candidates that hash to it, stored in sorted order. Each internal node actually contains a hash table with links to children nodes. Figure 6.6 shows one possible hash tree for candidate itemsets of size 3, which were shown

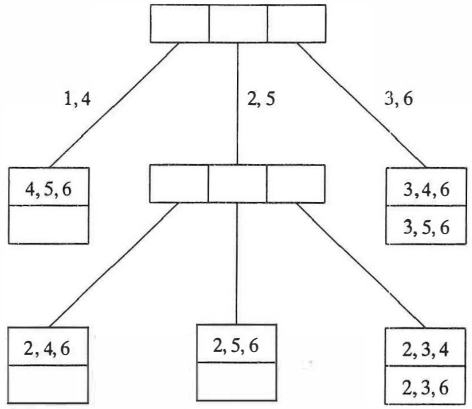


FIGURE 6.6: Hash tree for C_3 shown in Table 6.7.

in Table 6.7. For simplicity we have replaced each item with its numeric value in order: Blouse is 1, Jeans is 2, and so on. Here items 1, 4 hash to the first entry; 2, 5 hash to the second entry; and 3, 6 hash to the third entry.

- **Transaction data structure:** Transactions may be viewed as in a flat file or as a TID list, which can be viewed as an inverted file. The items usually are encoded (as seen in the hash tree example), and the use of bit maps has also been proposed.
- **Optimization:** These techniques look at how to improve on the performance of an algorithm given data distribution (skewness) or amount of main memory.
- **Architecture:** Sequential, parallel, and distributed algorithms have been proposed.
- **Parallelism strategy:** Both data parallelism and task parallelism have been used.

Table 6.8 (derived from [DXGH00]) provides a high-level comparison of the association rule algorithms we have covered in this chapter. When m is the number of items, the maximum number of scans is $m + 1$ for the level-wise algorithms. This applies to the parallel algorithms because they are based on Apriori. Both sampling algorithms and

TABLE 6.8: Comparison of Association Rule Algorithms (modified from [DXGH00])

Partitioning	Scans	Data Structure	Parallelism
Apriori	$m + 1$	hash tree	none
Sampling	2	not specified	none
Partitioning	2	hash table	none
CDA	$m + 1$	hash tree	data
DDA	$m + 1$	hash tree	task

partitioning algorithms require at most two complete scans of the transaction database. However, remember that the sampling algorithm must access the database to read the sample into memory and then many scans of it into memory may be required. Similarly, for the partitioning algorithm, each partition must be read into memory and may be scanned there multiple times.

6.6 INCREMENTAL RULES

All algorithms discussed so far assume a static database. However, in reality we cannot assume this. With these prior algorithms, generating association rules for a new database state requires a complete rerun of the algorithm. Several approaches have been proposed to address the issue of how to maintain the association rules as the underlying database changes. Most of the proposed approaches have addressed the issue of how to modify the association rules as inserts are performed on the database. These *incremental updating* approaches concentrate on determining the large itemsets for $D \cup db$ where D is a database state and db are updates to it and where the large itemsets for D, L are known.

One incremental approach, *fast update (FUP)*, is based on the Apriori algorithm. Each iteration, k, scans both db and D with candidates generated from the prior iteration, $k-1$, based on the large itemsets at that scan. In addition, we use as part of the candidate set for scan k to be L_k found in D. The difference is that the number of candidates examined at each iteration is reduced through pruning of the candidates. Although other pruning techniques are used, primary pruning is based on the fact that we already know L from D. Remember that according to the large itemset property, an itemset must be large in at least one of these partitions of the new database. For each scan k of db, L_k plus the counts for each itemset in L_k are used as input. When the count for each item in L_k is found in db, we automatically know whether it will be large in the entire database without scanning D. We need not even count any items in L_k during the scan of db if they have a subset that is not large in the entire database.

6.7 ADVANCED ASSOCIATION RULE TECHNIQUES

In this section we investigate several techniques that have been proposed to generate association rules that are more complex than the basic rules.

6.7.1 Generalized Association Rules

Using a concept hierarchy that shows the set relationship between different items, generalized association rules allow rules at different levels. Example 6.7 illustrates the use of these generalized rules using the concept hierarchy in Figure 6.7. Association rules could be generated for any and all levels in the hierarchy. A *generalized association rule*, $X \Rightarrow Y$, is defined like a regular association rule with the restriction that no item in Y may be above any item in X. When generating generalized association rules, all possible rules are generated using one or more given hierarchies. Several algorithms have been proposed to generate generalized rules. The simplest would be to expand each transaction by adding (for each item in it) all items above it in any hierarchy.

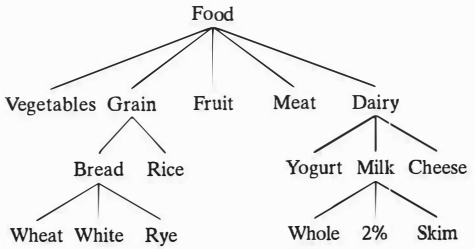


FIGURE 6.7: Concept hierarchy.

EXAMPLE 6.7

Figure 6.7 shows a partial concept hierarchy for food. This hierarchy shows that Wheat Bread is a type of Bread, which is a type of grain. An association rule of the form Bread $\Rightarrow$ PeanutButter has a lower support and threshold than one of the form Grain $\Rightarrow$ PeanutButter. There obviously are more transactions containing any type of grain than transactions containing Bread. Likewise, Wheat Bread $\Rightarrow$ Peanutbutter has a lower threshold and support than Bread $\Rightarrow$ PeanutButter.

6.7.2 Multiple-Level Association Rules

A variation of generalized rules are *multiple-level association rules*. With multiple-level rules, itemsets may occur from any level in the hierarchy. Using a variation of the Apriori algorithm, the concept hierarchy is traversed in a top-down manner and large itemsets are generated. When large itemsets are found at level i, large itemsets are generated for level $i+1$. Large k-itemsets at one level in the concept hierarchy are used as candidates to generate large k-itemsets for children at the next level.

Modification to the basic association rule ideas may be changed. We expect that there is more support for itemsets occurring at higher levels in the concept hierarchy. Thus, the minimum support required for association rules may vary based on level in the hierarchy. We would expect that the frequency of itemsets at higher levels is much greater than the frequency of itemsets at lower levels. Thus, for the reduced minimum support concept, the following rules apply:

- The minimum support for all nodes in the hierarchy at the same level is identical.
- If α_i is the minimum support for level i in the hierarchy and α_{i-1} is the minimum support for level $i-1$, then $\alpha_{i-1} > \alpha_i$.

6.7.3 Quantitative Association Rules

The association rule algorithms discussed so far assume that the data are categorical. A *quantitative association rule* is one that involves categorical and quantitative data. An

example of a quantitative rule is:

A customer buys wine for between \$30 and \$50 a bottle ⇒ she also buys caviar

This differs from a traditional association rule such as:

A customer buys wine ⇒ she also buys caviar.

The cost quantity has been divided into an interval (much as was done when we looked at handling numeric data in clustering and classification). In these cases, the items are not simple literals. For example, instead of having the items {Bread, Jelly}, we might have the items {(Bread:[0 . . . 1]), (Bread:(1 . . . 2]), (Bread:(2 . . . ∞)), (Jelly:[0 . . . 1.5]), (Jelly:(1.5 . . . 3]), (Jelly:(3 . . . ∞))}.

The basic approach to finding quantitative association rules is found in Algorithm 6.8. Here we show the Apriori algorithm being used to generate the large itemsets, but any such algorithm could be used.

ALGORITHM 6.8

```
Input:
   I       //Itemsets
   p^1, p^2, ..., p^p;          //Processors
   D = D^1, D^2, ..., D^p;          //Database divided into partitions
   s       //Support
Output:
   L       //Large itemsets
Quantitative association rule algorithm:
   for each I_j ∈ I do    //Partition items.
      if I_j is to be partitioned, then
         determine number of partitions;
         map attribute values into new partitions creating new items;
         replace I_j in I with the new items I_j1, ..., I_jm;
   Apriori(I, D, s);
```

Because we have divided what was one item into several items, the minimum support and confidence used for quantitative rules may need to be lowered. The minimum support problem obviously is worse with a large number of intervals. Thus, an alternative solution would be to combine adjacent intervals when calculating support. Similarly, when there are a small number of intervals, the confidence threshold may need to be lowered. For example, look at $X \Rightarrow Y$. Suppose there are only two intervals for X. Then the count for those transactions containing X will be quite high when compared to those containing Y (if this is a more typical item with many intervals).

6.7.4 Using Multiple Minimum Supports

When looking at large databases with many types of data, using one minimum support value can be a problem. Different items behave differently. It certainly is easier to obtain a given support threshold with an attribute that has only two values than it is with an attribute that has hundreds of values. It might be more meaningful to find a rule of the form

SkimMilk ⟹ WheatBread

with a support of 3% than it is to find

Milk ⟹ Bread

with a support of 6%. Thus, having only one support value for all association rules may not work well. Some useful rules could be missed. This is particularly of interest when looking at generalized association rules, but it may arise in other situations as well. Think of generating association rules from a non–market basket database. As was seen with quantitative rules, we may partition attribute values into ranges. Partitions that have a small number of values obviously will produce lower supports than those with a large number of values. If a larger support is used, we might miss out on generating meaningful association rules.

This problem is called the *rare item problem*. If the minimum support is too high, then rules involving items that rarely occur will not be generated. If it is set too low, then too many rules may be generated, many of which (particularly for the frequently occurring items) are not important. Different approaches have been proposed to handle this. One approach is to partition the data based on support and generate association rules for each partition separately. Alternatively, we could group rare items together and generate association rules for these groupings. A more recent approach to handling this problem is to combine clustering and association rules. First we cluster the data together based on some clustering criteria, and then we generate rules for each cluster separately. This is a generalization of the partitioning of the data solution.

One approach, *MISapriori*, allows a different support threshold to be indicated for each item. Here *MIS* stands for *minimum item support*. The minimum support for a rule is the minimum of all the minimum supports for each item in the rule. An interesting problem occurs when multiple minimum supports are used. The minimum support requirement for an itemset may be met even though it is not met for some of its subsets. This seems to violate the large itemset property. Example 6.8, which is adapted from [LHM99], illustrates this. A variation of the downward closure property, called the *sorted downward closure* property, is satisfied and used for the MISapriori algorithm. First the items are sorted in ascending MIS value. Then the candidate generation at scan 2 looks only at adding to a large item any item following it (larger than or equal to MIS value) in the sorted order.

EXAMPLE 6.8

Suppose we have three items, $\{A, B, C\}$, with minimum supports $MIS(A) = 20\%$, $MIS(B) = 3\%$, and $MIS(C) = 4\%$. Because the support for A is so large, it may be small, while both AB and AC may be large because the required support for $AB = \min(MIS(A), MIS(B)) = 3\%$ and $AC = \min(MIS(A), MIS(C)) = 4\%$.

6.7.5 Correlation Rules

A *correlation rule* is defined as a set of itemsets that are correlated. The motivation for developing these correlation rules is that negative correlations may be useful.

Example 6.9, which is modified from [BMS77], illustrates this concept. In this example, even though the probability of purchasing two items together seems high, it is much higher if each item is purchased without the other item. Correlation satisfies upward closure in the itemset lattice. Thus, if a set is correlated, so is every superset of it.

EXAMPLE 6.9

Suppose there are two items, $\{A, B\}$ where $A \Rightarrow B$ has a support of 15% and a confidence of 60%. Because these values are high, a typical association rule algorithm probably would deduce this to be a valuable rule. However, if the probability to purchase item B is 70%, then we see that the probability of purchasing B has actually gone down, presumably because A was purchased. Thus, there appears to be a negative correlation between buying A and buying B. The correlation can be expresed as

$$\text{correlation}(A \Longrightarrow B) = \frac{P(A, B)}{P(A)\ P(B)} \tag{6.1}$$

which in this case is: $\frac{0.15}{0.25 \times 0.7} = 0.857$. Because this correlation value is lower than 1, it indicates a negative correlation between A and B.

6.8 MEASURING THE QUALITY OF RULES

Support and confidence are the normal methods used to measure the quality of an association rule:

$$s(A \Longrightarrow B) = P(A, B) \tag{6.2}$$

and

$$\alpha(A \Longrightarrow B) = P(B \mid A) \tag{6.3}$$

However, there are some problems associated with these metrics. For example, confidence totally ignores $P(B)$. A rule may have a high support and confidence but may be an obvious rule. For example, if someone purchases potato chips, there may be a high likelihood that he or she would also buy a cola. This rule is not really of interest because it is not surprising. Various concepts such as surprise and interest have been used to evaluate the quality or usefulness of rules. We briefly examine some of these in this section.

With correlation rules, we saw that correlation may be used to measure the relationship between items in a rule. This may also be expressed as the *lift* or *interest*

$$\text{interest}(A \Longrightarrow B) = \frac{P(A, B)}{P(A)\ P(B)} \tag{6.4}$$

This measure takes into account both $P(A)$ and $P(B)$. A problem with this measure is that it is symmetric. Thus, there is no difference between the value for interest($A \Rightarrow B$) and the value for interest($B \Rightarrow A$).

As with lift, *conviction* takes into account both $P(A)$ and $P(B)$. From logic we know that implication $A \rightarrow B \equiv \neg(A \wedge \neg B)$. A measure of the independence of the negation of implication, then, is $\frac{P(A,\neg B)}{P(A)\ P(\neg B)}$. To take into account the negation, the conviction measure inverts this ratio. The formula for *conviction* is [BMS77]

$$\text{conviction}(A \Longrightarrow B) = \frac{P(A)\ P(\neg B)}{P(A, \neg B)} \tag{6.5}$$

Conviction has a value of 1 if A and B are not related. Rules that always hold have a value of ∞.

The usefulness of discovered association rules may be tied to the amount of surprise associated with the rules or how they deviate from previously known rules. Here *surprise* is a measure of the changes of correlations between items over time. For example, if you are aware that beer and pretzels are often purchased together, it would be a surprise if this relationship actually lowered significantly. Thus, this rule beer $\Rightarrow$ pretzel would be of interest even if the confidence decreased.

Another technique to measure the significance of rules by using the chi squared test for independence has been proposed. This significance test was proposed for use with correlation rules. Unlike the support or confidence measurement, the chi squared significance test takes into account both the presence and the absence of items in sets. Here it is used to measure how much an itemset (potential correlation rule) count differs from the expected. The chi squared test is well understood because it has been used in the statistics community for quite a while. Unlike support and confidence, where arbitrary values must be chosen to determine which rules are of interest, the chi squared values are well understood with existing tables that show the critical values to be used to determine relationships between items.

The chi squared statistic can be calculated in the following manner. Suppose the set of items is $I = \{I_1, I_2, \ldots, I_m\}$. Because we are interested in both the occurrence and the nonoccurrence of an item, a transaction t_j can be viewed as

$$t_j \in \{I_1, \bar{I}_1\} \times \{I_2, \bar{I}_2\} \times \cdots \times \{I_m, \bar{I}_m\} \tag{6.6}$$

Given any possible itemset X, it also is viewed as a subset of the Cartesian product. The chi squared statistic is then calculated for X as

$$\chi^2 = \sum_{X \in I} \frac{(O(X) - E[X])^2}{E[X]} \tag{6.7}$$

Here $O(X)$ is the count of the number of transactions that contain the items in X. For one item I_i, the expected value is $E[I_i] = O(I_i)$, the count of the number of transactions that contain I_i. $E[\bar{I}_i] = n - O(I_i)$. The expected value $E[X]$ is calculated assuming independence and is thus defined as

$$E[X] = n \times \prod_{i=1}^{m} \frac{E[I_i]}{n} \tag{6.8}$$

Here n is the number of transactions.

Table 6.9, which is called a *contingency table*, shows the distribution of the data in Example 6.9 assuming that the sample has 100 items in it. From this we find

TABLE 6.9: Contingency Table for Example 6.9

	B	$\bar{B}$	Total
A	15	10	25
$\bar{A}$	55	20	75
Total	70	30	100

$E[AB] = 17.5$, $E[A\bar{B}] = 7.5$, $E[\bar{A}B] = 52.5$, and $E[\bar{A}\bar{B}] = 22.5$. Using these values, we calculate χ^2 for this example as

$$\chi^2 = \sum_{X \in I} \frac{(O(X) - E[X])^2}{E[X]} = \frac{(15 - 17.5)^2}{17.5} + \frac{(10 - 7.5)^2}{7.5} + \frac{(55 - 52.5)^2}{52.5} + \frac{(20 - 22.5)^2}{22.5} = 1.587 \quad (6.9)$$

If all values were independent, then the chi squared statistic should be 0. A chi squared table (found in most statistics books) can be examined to interpret this value. Examining a table of critical values for the chi squared statistic, we see that a chi squared value less than 3.84 indicates that we should not reject the independent assumption. This is done with 95% confidence. Thus, even though there appears to be a negative correlation between A and B, it is not statistically significant.

6.9 EXERCISES

1. Trace the results of using the Apriori algorithm on the grocery store example with $s = 20\%$ and $\alpha = 40\%$. Be sure to show the candidate and large itemsets for each database scan. Also indicate the association rules that will be generated.
2. Prove that all potentially large itemsets are found by the repeated application of BD^- as is used in the sampling algorithm.
3. Trace the results of using the sampling algorithm on the clothing store example with $s = 20\%$ and $\alpha = 40\%$. Be sure to show the use of the negative border function as well as the candidates and large itemsets for each database scan.
4. Trace the results of using the partition algorithm on the grocery store example with $s = 20\%$ and $\alpha = 40\%$. For the grocery store example, use two partitions of size 2 and 3, respectively. You need not show all the steps involved in finding the large itemsets for each partition. Simply show the resulting large itemsets found for each partition.
5. Trace the results of using the count distribution algorithm on the clothing data with $s = 20\%$. Assume that there are three processors with partitions created from the beginning of the database of size 7, 7, and 6, respectively.
6. Trace the results of using the data distribution algorithm on the clothing data with $s = 20\%$. Assume that there are three processors with partitions created from the beginning of the database of size 7, 7, and 6, respectively. Assume that candidates are distributed at each scan by dividing the total set into subsets of equal size.
7. Calculate the lift and conviction for the rules shown in Table 6.3. Compare these to the shown support and confidence.
8. Perform a survey of recent research examining techniques to generate rules incrementally.

6.10 BIBLIOGRAPHIC NOTES

The development of association rules can be traced to one paper in 1993 [AIS93]. Agrawal proposed the AIS algorithm before *Apriori* [AIS93]. However, this algorithm and another, SETM [HS95], do not take advantage of the large itemset property and thus generate too many candidate sets. The a priori algorithm is still the major technique used by commercial products to detect large itemsets. It was proposed by 1994 in [AS94]. Another algorithm proposed about the same time, OCD, uses sampling [MTV94]. It produces fewer candidates than AIS.

There have been many proposed algorithms that improve on Apriori. Apriori-TID does not use the database to count support [AS94]. Instead, it uses a special encoding fo the candidates from the previous pass. Apriori has better performance in early passes of the database while Apriori-*TID* has better performance in later. A combination of the two, Apriori-Hybrid, has been proposed [Sri96]. The dynamic itemset counting (DIC) algorithm divides the database into intervals (like the partitions in the partition algorithm) [BMUT77]. The scan of first interval counts the 1-itemsets. Then candidates of size 1 are generated. The scan of the second interval, then, counts the 1-itemsets as well as those 2-itemsets. In this manner itemsets are counted earlier. However, more memory space may be required. The partition algorithm was first studied in 1995 by Savasere [SON95], while the sampling algorithm is attributed to Toivonen in 1996 [Toi96]. The problem of uneven distribution of data in the partition algorithm was addressed in [LD98], where a set of algorithms were proposed that better prune away false candidates before the second scan.

The CDA and DDA algorithms were both proposed in [AS96]. Other data parallel algorithms include PDM [PCY95], DMA [CHN+96], and CCPD [ZOPL96]. Additional task parallel algorithms include IDD [HKK97], HPA [SK96], and PAR [ZPOL97]. A hybrid approach, hybrid distribution (HD), which combines the advantages of each technique has a speed-up close to the data parallelism approach [HKK97]. For a discussion of other parallel algorithms, see either [DXGH00] or [Zak99].

Many additional algorithms have been proposed. CARMA (continuous association rule mining algorithm) [Hid99] proposes a technique that is dynamic in that it allows the user to change the support and confidence while the algorithm is running. Some recent work has examined the use of an AI type search algorithm called *OPUS* [Web00]. OPUS prunes out portions of the search tree based on the desired rule characteristics. However, many scans of the database are required and, thus it assumes that the database is memory-resident.

An approach to determining if an item should be partitioned when generating quantitative rules has been proposed [SA96a]. Variations of quantitative rules include *profile association rules* where the left side of the rule represents some profile information