4

High-Dimensional Data

Chapters 1 and 2 dealt with the representation of multidimensional points and objects, respectively, and the development of appropriate indexing methods that enable them to be retrieved efficiently. Most of these methods were designed for use in application domains where the data usually has a spatial component that has a relatively low dimension. Examples of such application domains include geographic information systems (GIS), spatial databases, solid modeling, computer vision, computational geometry, and robotics. However, there are many application domains where the data is of considerably higher dimensionality and is not necessarily spatial. This is especially true in pattern recognition and image databases where the data is made up of a set of objects, and the high dimensionality is a direct result of trying to describe the objects via a collection of features (also known as a *feature vector*). Examples of features include color, color moments, textures, shape descriptions, and so on, expressed using scalar values. The goal in these applications is often one of the following:

1. Finding objects having particular feature values (point queries).
2. Finding objects whose feature values fall within a given range or where the distance from some query object falls into a certain range (range queries).[1]
3. Finding objects whose features have values similar to those of a given query object or set of query objects (nearest neighbor queries). In order to reduce the complexity of the search process, the precision of the required similarity can be an approximation (approximate nearest neighbor queries).
4. Finding pairs of objects from the same set or different sets that are sufficiently similar to each other (all-closest-pairs queries). This is also a variant of a more general query commonly known as a *spatial join query*.

These queries are collectively referred to as *similarity searching* (also known as *similarity retrieval*), and supporting them is the subject of this chapter. The objective of the search can either be to find a few similar objects from a larger set or to match corresponding objects in two or more large sets of objects so that the result is a pairing of all of the objects from one set with objects in the other set. When the pairing is one-to-one (i.e., where for each distinct object in set A, we seek one nearest neighboring object in set B, which need not always be distinct), then the latter query is known as the *all nearest neighbors problem*. There is also the general problem (e.g., [226, 2030, 2066]), where B need not be the same as A, while in [297, 396, 1908], B is constrained to be

[1] When we are only interested in the number of objects that fall within the range, instead of the identity of the objects, then the query is known as a *range aggregation query* (e.g., [1853, 1203]).

the same as A),[2] and the distance semijoin for $k=1$ [847]. An example of the utility of the latter is the *conflation problem*, which arises often in applications involving spatial data in GIS. In its simplest formulation, conflation is graph or network matching and is usually applied in a geographic setting to maps that are slightly out of alignment in a spatial sense (e.g., [155, 248, 344, 1598, 1599, 1601, 1958, 2033]). We do not deal with such searches in this book and, instead, focus on the former. Moreover, of the queries enumerated above, the nearest neighbor query also known as the *post office problem* (e.g., [1046, p, 563]), is particularly important and hence it is emphasized here. This problem arises in many different fields including computer graphics, where it is known as a *pick query* (e.g., [622]); in coding, where it is known as the *vector quantization problem* (e.g., [747]); and in pattern recognition, as well as machine learning, where it is known as the *fast nearest-neighbor classifier* (e.g., [514]). An apparently straightforward solution to finding the nearest neighbor is to compute a Voronoi diagram for the data points (i.e., a partition of the space into regions where all points in the region are closer to the region's associated data point than to any other data point) and then locate the Voronoi region corresponding to the query point. As we will see in Section 4.4.4, the problem with this solution is that the combinatorial complexity of the Voronoi diagram in high dimensions, expressed in terms of the number of objects, is prohibitive, thereby making it virtually impossible to store the structure, which renders its applicability moot. However, see the approximate Voronoi diagram (AVD) in Section 4.4.5, where the dimension of the underlying space is captured by expressing the complexity bounds in terms of the error threshold ε rather than the number of objects N in the underlying space.

The above is typical of the problems that we must face when dealing with high-dimensional data. Generally speaking, multidimensional problems such as these queries become increasingly more difficult to solve as the dimensionality increases. One reason is that most of us are not particularly adept at visualizing high-dimensional data (e.g., in three and higher dimensions, we no longer have the aid of paper and pencil). However, more importantly, we eventually run into the *curse of dimensionality*. This term was coined by Bellman [159] to indicate that the number of samples needed to estimate an arbitrary function with a given level of accuracy grows exponentially with the number of variables (i.e., dimensions) that it comprises. For similarity searching (i.e., finding nearest neighbors), this means that the number of objects (i.e., points) in the dataset that need to be examined in deriving the estimate grows exponentially with the underlying dimension.

The curse of dimensionality has a direct bearing on similarity searching in high dimensions in the sense that it raises the issue of whether or not nearest neighbor searching is even meaningful in such a domain. In particular, letting d denote a distance function that need not necessarily be a metric, Beyer, Goldstein, Ramakrishnan, and Shaft [212] point out that nearest neighbor searching is not meaningful when the ratio of the variance of the distance between two random points p and q, drawn from the data and query distributions, and the expected distance between them converges to zero as the dimension k goes to infinity—that is,

$$\lim_{k\to\infty} \frac{\text{Variance}[d(p,q)]}{\text{Expected}[d(p,q)]} = 0.$$

In other words, the distance to the nearest neighbor and the distance to the farthest neighbor tend to converge as the dimension increases. Formally, they prove that when the data and query distributions satisfy this ratio, then the probability that the farthest neighbor distance is smaller than $1+\varepsilon$ of the nearest neighbor distance is 1 in the limit as the dimension goes to infinity and ε is a positive value. For example, they show that this ratio holds whenever the coordinate values of the data and the query point are independent

[2] Some of the solutions (e.g., [226, 1908, 2030]) try to take advantage of the fact that the nearest neighbor search needs to be performed on all points in the dataset and thus try to reuse the results of efforts expended in prior searches. They arise primarily in the context of database applications, where they are known as an *all nearest neighbor join* (ANN join).

and identically distributed, as is the case when they are both drawn from a uniform distribution. This is easy to see in the unit hypercube as the variance is smaller than the expected distance.

Assuming that d is a distance metric and hence that the triangle inequality holds, an alternative way of looking at the curse of dimensionality is to observe that, when dealing with high-dimensional data, the probability density function (analogous to a histogram) of the distances of the various elements is more concentrated and has a larger mean value. This means that similarity searching algorithms will have to perform more work. In the worst case, we have the situation where $d(x,x)=0$ and $d(y,x)=1$ for all $y \neq x$, which means that a similarity query must compare the query object with every object of the set. One way to see why more concentrated probability densities lead to more complex similarity searching is to observe that this means that the triangle inequality cannot be used so often to eliminate objects from consideration. In particular, the triangle inequality implies that every element x such that $|d(p,q)-d(p,x)| > \varepsilon$ cannot be at a distance of ε or less from q (i.e., from $d(p,q) \le d(p,x)+d(q,x)$). Thus, if we examine the probability density function of $d(p,x)$ (i.e., on the horizontal axis), we find that when ε is small while the probability density function is large at $d(p,q)$, then the probability of eliminating an element from consideration via the use of the triangle inequality is the remaining area under the curve, which is quite small (see Figure 4.1(a) in contrast to Figure 4.1(b) where the density function of the distances is more uniform). This is not surprising because the sum of the area under the curve corresponding to the probability density function is 1. Note that use of the triangle inequality to eliminate objects from consideration is analogous to the application of the method of Friedman, Baskett, and Shustek [649] as well as Nene and Nayar [1364] for vector spaces, which eliminates a k-dimensional object $x=(x_0,x_1,\ldots,x_{k-1})$ from consideration as being within ε of $q=(q_0,q_1,\ldots,q_{k-1})$ if $|x_i-q_i| > \varepsilon$ for one of x_i where $0 \le i \le k-1$ (see Section 1.1 in Chapter 1).

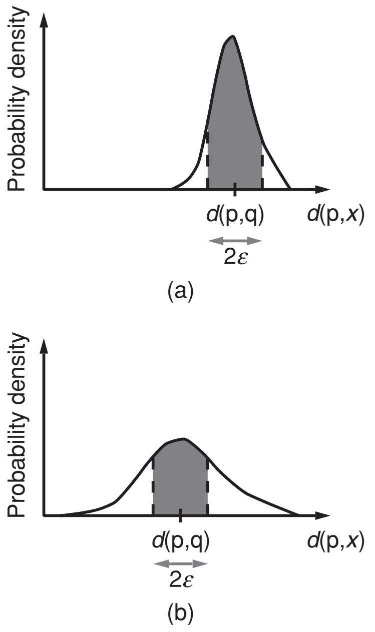

Figure 4.1
A probability density function (analogous to a histogram) of the distances $d(\mathrm{p},x)$ with the shaded area corresponding to $|d(\mathrm{p},\mathrm{q})-d(\mathrm{p},x)| \le \varepsilon$: (a) a density function where the distance values have a small variation; (b) a more uniform distribution of distance values, thereby resulting in a more effective use of the triangle inequality to prune objects from consideration as satisfying the range search query.

These observations mean that nearest neighbor searching may be quite inefficient as it is very difficult to differentiate between the nearest neighbor and the other elements. Moreover, as we will see, seemingly appropriate indexing methods, analogous to those described in Chapters 1 and 2, which are designed to make it easier to avoid examining irrelevant elements, may not be of great help in this case. In fact, the experiments of Beyer et al. [212] show that the curse of dimensionality becomes noticeable for dimensions as low as 10–15 for the uniform distribution. The only saving grace is that real-world high-dimensional data (say of dimension k) is not likely to be uniformly distributed as its volume is much smaller than $O(c^k)$ for some small constant $c > 2$. Thus, we can go on with our discussion despite the apparent pall of the curse of dimensionality, which tends to cast a shadow on any arguments or analyses that are based on uniformly distributed data or queries.

Assuming that the curse of dimensionality does not come into play, query responses are facilitated by sorting the objects on the basis of some of their feature values and building appropriate indexes. The high-dimensional feature space is indexed using some multidimensional data structure (termed *multidimensional indexing*) such as those described in Chapters 1 and 2 or modifications thereof to fit the high-dimensional problem environment. Similarity search that finds objects similar to a target object can be done with a range search or a nearest neighbor search in the multidimensional data structure. However, unlike applications in spatial databases where the distance function between two objects is usually Euclidean, this is not necessarily the case in the high-dimensional feature space where the distance function may even vary from query to query on the same feature (e.g., [1591]). Unless stated otherwise, we usually assume that the distance is measured directly between points in the space (i.e., "as the crow flies" in a spatial context) rather than possibly being constrained to be along a network on which the points lie, as is useful in applications such as spatial databases. Nevertheless, we do cover the situation that distance is measured along a spatial network in Section 4.1.6. We also assume that the data and query objects are static rather than in motion (e.g., [162, 296, 889, 930, 931, 949, 1306, 1320, 1543, 1789, 1855, 1851, 2034, 2061], many of which make use of the TPR-tree [1610] and the TPR*-tree [1855], which are variants of an R-tree [791] and an R*-tree [152], respectively).

Searching in high-dimensional spaces is time-consuming. Performing point and range queries in high dimensions is considerably easier, from the standpoint of computational complexity, than performing similarity queries because point and range queries do not involve the computation of distance. In particular, searches through an indexed space usually involve relatively simple comparison tests. However, if we have to examine all of the index nodes, then the process is again time-consuming. In contrast, computing similarity makes use of distance, and the process of computing the distance can be computationally complex. For example, computing the Euclidean distance between two points in a high-dimensional space, say d, requires d multiplication operations and $d-1$ addition operations, as well as a square root operation (which can be omitted). Note also that computing similarity requires the definition of what it means for two objects to be similar, which is not always so obvious.

The previous discussion has been based on the premise that we know the features that describe the objects (and hence the dimensionality of the underlying feature space). In fact, it is often quite difficult to identify the features, and thus we frequently turn to experts in the application domain from which the objects are drawn for assistance in this process. Nevertheless, it is often the case that the features cannot be easily identified even by the domain experts. In this case, the only information that we have available is a distance function that indicates the degree of similarity (or dissimilarity) between all pairs of objects, given a set of N objects. Usually, it is required that the distance function d obey the triangle inequality, be nonnegative, and be symmetric, in which case it is known as a *metric* and also referred to as a *distance metric* (see Section 4.5.1). Sometimes, the degree of similarity is expressed by use of a similarity matrix that contains interobject distance values for all possible pairs of the N objects. Some examples of distance functions that are distance metrics include edit distances, such as the Levenshtein [1156] and Hamming [796] distances for strings[3] and the Hausdorff distance for images (e.g., [911] and footnote 45 in Section 2.2.3.4 of Chapter 2).

Given a distance function, we usually index the data (i.e., objects) with respect to their distance from a few selected objects. We use the term *distance-based indexing* to describe such methods. The advantage of distance-based indexing methods is that distance computations are used to build the index, but once the index has been built, similarity queries can often be performed with a significantly lower number of distance computations than a sequential scan of the entire dataset. Of course, in situations where we may want to apply several different distance metrics, then the drawback of the distance-based indexing techniques is that they require that the index be rebuilt for each different distance metric, which may be nontrivial. This is not the case for the multidimensional indexing methods that have the advantage of supporting arbitrary distance metrics (however, this comparison is not entirely fair since the assumption, when using distance-based indexing, is that often we do not have any feature values, as for example in DNA sequences).

There are many problems with indexing high-dimensional data. In fact, due to the complexity of handling high-dimensional data using a multidimensional index, we frequently find that the cost of performing queries using the index is higher than a sequential scan of the entire data (e.g., [212]). This is a result of the curse of dimensionality, which was discussed earlier. However, the “inherent dimensionality” of a dataset is often much lower than the dimensionality of the underlying space. For example, the values of some of the features may be correlated in some way. Alternatively, some of the features may

[3] The *Levenshtein edit distance* between two strings s and t is the number of deletions, insertions, or substitutions required to transform s into t. The *Hamming edit distance* between s and t, defined only when s and t are of the same length, is the number of positions in which s and t differ (i.e., have different characters).

not be as important as others in discriminating between objects and thus may be ignored or given less weight (e.g., [838]). Therefore, there has been a considerable amount of interest in techniques to reduce the dimensionality of the data. Another motivation for the development of many dimension reduction techniques has been a desire to make use of disk-based spatial indexes that are based on object hierarchies, such as members of the R-tree family [791]. The performance of these methods decreases with an increase in dimensionality due to the decrease in the fanout of a node of a given capacity since usually the amount of storage needed for the bounding boxes is directly proportional to the dimensionality of the data, thereby resulting in longer search paths.

In situations where no features but only a distance function are defined for the objects, there exists an alternative to using distance-based indexes. In particular, methods have been devised for deriving "features" purely based on the interobject distances [587, 886, 1181, 1950]. Thus, given N objects, the goal is to choose a value of k and find a set of N corresponding points in a k-dimensional space so that the distance between the N corresponding points is as close as possible to that given by the distance function for the N objects. The attractiveness of such methods is that we can now index the points using multidimensional data structures. These methods are known as *embedding methods*[4] and can also be applied to objects represented by feature vectors as alternatives to the traditional dimension reduction methods. In fact, one of these methods [587] is inspired by dimension reduction methods based on linear transformations. Embedding methods can also be applied when the features are known, in which case they can also be characterized as dimension reduction techniques (e.g., [924]).

The rest of this chapter is organized as follows. Since our main motivation is similarity searching, we start with a description of the process of finding nearest neighbors in Section 4.1. Here the focus is on finding the nearest neighbor rather than the k nearest neighbors. Once the nearest neighbor has been found, the process that we describe can be used to find the second nearest neighbor, the third, and so on. This process is incremental, in contrast to an alternative method that, given an integer k, finds the k nearest neighbors regardless of the order of their distance from the query object q. The advantage of this approach over the incremental method is that the amount of storage is bounded by k, in contrast to having possibly to keep track of all of the objects if their distance from q is approximately the same. On the other hand, the advantage of the incremental approach is that once the k nearest neighbors have been obtained, the $(k+1)$-th neighbor can be obtained without having to start the process anew to obtain $k+1$ nearest neighbors.

Note that the problem of finding neighbors is somewhat similar to the point location query, which was discussed in Section 2.1.3. In particular, in many algorithms, the first step is to locate the query object q. The second step consists of the actual search. Most of the classical methods employ a "depth-first" *branch and bound* strategy (e.g., [836]). An alternative, and the one we describe in Section 4.1, employs a "best-first" search strategy. The difference between the two strategies is that, in the best-first strategy, the elements of the structure in which the objects are stored are explored in increasing order of their distance from q. In contrast, in the depth-first strategy, the order in which the elements of the structure in which the objects are stored are explored is a result of performing a depth-first traversal of the structure using the distance to the currently nearest object to prune the search. The depth-first approach is exemplified by the algorithm of Fukunaga and Narendra [666] and is discussed in Section 4.2, where it is also compared with the general incremental algorithm described in Section 4.1. During this comparison, we also show how to use the best-first strategy when only the k nearest neighbors are desired.

The best-first process of finding nearest neighbors incrementally yields a ranking of the objects with respect to their distance from the query object. This query has taken on an importance of its own. For example, it forms the basis of the *distance join* [847,

[4] These methods are distinct and unrelated to embedding space organization methods discussed in the opening text of Chapter 1 and throughout that chapter.

1756] query, where each element of A is paired up with elements in B, and the results are obtained in increasing order of distance. Another variant of this query is the *distance semijoin*, which arises when each element a in A can be paired up with just one element b in B, and we stipulate that b is the closest element in B to a, as well as report the results in increasing order of distance. This query is executed in a coroutine manner, obtaining as many results as necessary. When the query is executed by applying to A in its entirety, thereby letting the number of neighbors $k = |A|$, the result is equivalent to a discrete Voronoi diagram on B [847]. Moreover, by repeated application of the distance semijoin or distance join to the elements of one set A with respect to a second set B, and letting $k = 1$, the results of this query can be used as part of a process to find closest pairs (i.e., the all nearest neighbor join discussed earlier). The incremental algorithm finds use in many applications where the number of neighbors that are needed is not always known before starting the query (e.g., in forming query execution plans for query optimization in database applications).

In situations where complete accuracy is not critical, we can make use of the notion of approximate nearest neighbors. We explore two possible approaches to the problem. In the conventional approach, the approximation is the resulting quantitative error, which can be measured (Section 4.3). On the other hand, in the second, more unusual, approach, the approximation results from the assumption that "nearest neighbor" is an "equivalence" relation (Section 4.5.8), which, of course, is not generally true. For the first approach, we demonstrate how the performance of both the best-first and depth-first algorithms can be improved at the risk of having the objects reported somewhat out of order.

The presentation continues in Section 4.4, which presents some multidimensional indexing methods. Many of these methods are high-dimensional adaptations of techniques presented in Chapters 1 and 2. Section 4.5 outlines some distance-based indexing methods. Section 4.6 discusses a number of dimension reduction techniques and shows how some of them are used in conjunction with some of the indexing techniques presented in Chapters 1 and 2. Section 4.7 concludes by presenting a number of embedding methods. Note that many of the embedding methods (e.g., locality sensitive hashing [924]) could also have been discussed in the context of dimension reduction techniques. Our decision on the placement of the discussion depended on whether the emphasis was on the extent of the dimension reduction or on the satisfaction of other properties of the embedding.

4.1 Best-First Nearest Neighbor Finding

In this section, we present a general algorithm for finding nearest neighbors. Rather than finding the k nearest neighbors and possibly having to restart the search process should more neighbors be needed, our focus is on finding them in an incremental manner. The nature of such search is to report the objects in a dataset $S \subset \mathbb{U}$, one by one, in order of distance from a query object $q \in \mathbb{U}$ based on a distance function d, where $\mathbb{U}$ is the domain (usually infinite) from which the objects are drawn. The algorithm is a general version of the incremental nearest neighbor algorithm of [846, 848] that is applicable to virtually all hierarchical indexing methods for both spatial (point data and objects with extent) and metric data, in which case the index is based only on the distance between the objects (see Section 4.5). In order to make the discussion more concrete, we use the R-tree (recall Section 2.1.5.2 of Chapter 2) as an example data structure.

The rest of this section is organized as follows: In Section 4.1.1, we motivate the incremental nearest neighbor problem by examining some of the issues and proposed solutions. In Section 4.1.2, we introduce the basic framework for performing search in the form of a *search hierarchy*. In Section 4.1.3, we present a general incremental nearest neighbor algorithm, based on the abstract concept of a search hierarchy, as well as a discussion of its correctness. This algorithm can be adapted to virtually any data

Exercises

1. Given a probably correct os-tree where f holds for the leaf nodes, prove that if the cover of each node contains the covers of its children, then f is the failure probability for search in the tree.

2. When constructing the probably correct os-tree, once the points in S_a have been partitioned with the hyperplane H_a resulting from the use of the PCA method, it was proposed to find a more appropriate separating hyperplane I_a that minimizes the number of training points on the right side of I_a whose nearest neighbors are elements of S_{al} and the number of training points on the left side of I_a whose nearest neighbors are elements of S_{ar}. Would it not be more appropriate to choose the separating hyperplane I_a that minimizes the d-dimensional volume spanned by the training points on the right side of I_a whose nearest neighbors are elements of S_{al} and the volume spanned by the training points on the left side of I_a whose nearest neighbors are elements of S_{ar}? If yes, what are the drawbacks of such a choice?

3. When constructing the probably correct os-tree, what is the rationale for choosing the hyperplane that is farthest from the closest points on its two sides rather than one that is closer to the same two points?

4.4.5 Approximate Voronoi Diagram (AVD)

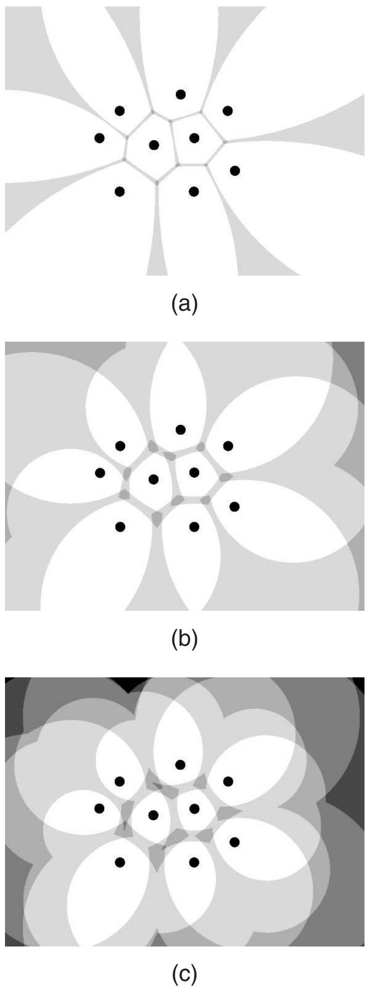

Figure 4.31
Partitions of the underlying space induced by the ε-nearest neighbor sets corresponding to the sites of the Voronoi diagram given in Figure 2.110(a) in Section 2.2.1.4 of Chapter 2 for (a) ε=0.10, (b) ε=0.30, and (c) ε=0.50. The darkness of the shading indicates the cardinality of the ε-nearest neighbor sets, with white corresponding to 1.

Har-Peled [799] addresses the $\Theta(N^{d/2})$ space requirements of the d-dimensional Voronoi diagram of a point set S by approximating it with an implicit representation that he terms an *approximate Voronoi diagram* (AVD). The idea is to partition the underlying space using some arbitrary block decomposition rule so that, given a $\varepsilon \geq 0$, every block b is associated with some element r_b in S such that r_b is an ε-nearest neighbor for all of the points in b. The motivation for the AVD is to reduce the space requirements for a d-dimensional Voronoi diagram from $\Theta(N^{d/2})$ for N points to a quantity closer to linear, although not necessarily linear. In particular, Har-Peled [799] shows that, for a given value of ε, it is possible to construct an AVD in $O((N/\varepsilon^d)(\log N)(\log(N/\varepsilon)))$ time, taking up the same amount of space (i.e., the number of blocks is also $O((N/\varepsilon^d)(\log N)(\log(N/\varepsilon))))$ and to determine the ε-nearest neighbor of a query point q in $O(\log(N/\varepsilon))$ time.

Note that the Voronoi diagram is only implicitly represented in the AVD in the sense that the boundaries of the Voronoi regions are not explicitly stored in the blocks. This implicit representation is also a characteristic of using the mb-tree [1383] (see Section 4.5.3.3) to represent data drawn from either a vector or a metric space where the data are objects, termed *pivots*, with associated regions so that all objects that are associated with a pivot p are closer to p than to any other pivot. In this case, the boundaries of the regions are also represented implicitly (see Section 4.5.3.3 for more details).

There are many possible block decomposition rules. Regardless of the rule that is chosen for each block, the only requirement is that the intersection of the ε-nearest neighbor sets of all of the points in each block be nonempty. For example, Figure 4.31(a–c), corresponds to the partitions of the underlying space induced by the ε-nearest neighbor sets corresponding to the sites of the Voronoi diagram given in Figure 2.110(a) in Section 2.2.1.4 of Chapter 2 for $\varepsilon = 0.10, 0.30$, and 0.50, respectively. The darkness of the shading indicates the cardinality of the ε-nearest neighbor sets, with white corresponding to 1. The space requirements and the time complexity of nearest neighbor queries are reduced when the block decomposition rule yields blocks that are sufficiently "fat" (i.e., they have a good aspect ratio as discussed in Section 1.5.1.4 of Chapter 1). One possibility, for which the "fat" requirement holds, is to use a rule, such as the PR quadtree [1413, 1637] (see Section 1.4.2.2), that for multidimensional point data recursively decomposes the underlying space into congruent blocks (i.e., squares in two dimensions) until each block is either empty or contains at most one point.

vertex with corresponding object o has an edge to each of the vertices that correspond to the k nearest neighbors of o. As an example, consider the distance matrix representing the interobject distances for the 18 objects a–r in Figure 4.63 discussed in Section 4.5.5.1. Recall that Figure 4.64(a) is the result of embedding these objects in the two-dimensional plane, assuming that the distance is the Euclidean distance and assigning them a consistent set of coordinate values so that these distance values hold. In addition, Figure 4.64(b) is the corresponding Delaunay graph, which for this particular planar embedding is a plane graph known as the Delaunay triangulation. Figure 4.67(a–e) shows the kNN graphs for $k = 1\ldots 5$, respectively, corresponding to the data in Figure 4.63 using the planar embedding depicted in Figure 4.64. Edges in the graphs that are shown as undirected correspond to bidirectional edges, which means that both vertices are in the k nearest neighbor sets of each other.

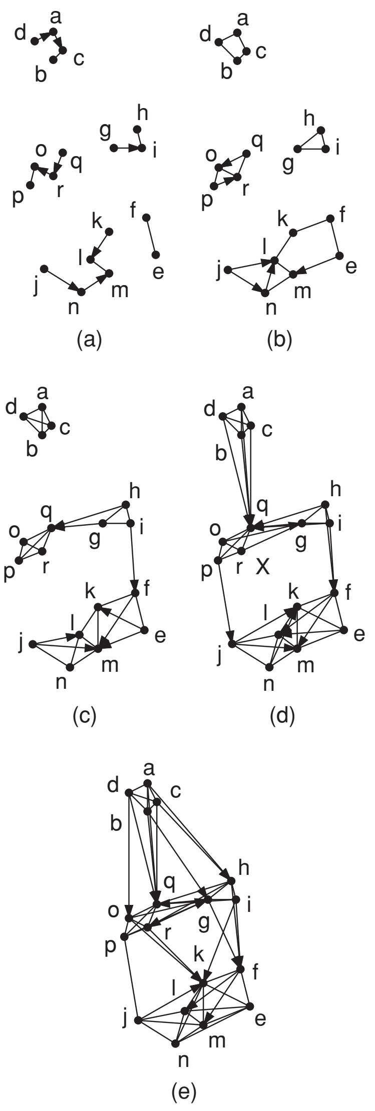

Figure 4.67
The kNN graphs for $k = 1\ldots 5$ corresponding to the data in Figure 4.63 using the planar embedding depicted in Figure 4.64(a): (a) $k = 1$, (b) $k = 2$, (c) $k = 3$, (d) $k = 4$, and (e) $k = 5$. Edges that are shown undirected correspond to bidirectional edges, which means that both vertices are in the k-nearest neighbor sets of each other.

Finding the nearest neighbor in S ($S \subset \mathbb{U}$) of query object q in $\mathbb{U}$ using the kNN graph can be achieved by using the same algorithm used in the Delaunay graph. In particular, start at an arbitrary object in S and proceed to a neighboring object in S that is closer to q as long as this is possible. Upon reaching an object o in S where the objects in its neighbor set $N(o)$ in S (i.e., the objects connected to o by an edge) are all farther away from q than o, we know that o is the nearest neighbor of q. Unfortunately, as can be seen from the different kNN graphs in Figure 4.67, there are situations where this algorithm will fail to yield the nearest neighbor. In particular, just because we found an object p whose k nearest neighbors are farther from the query object q than p does not necessarily mean that p is q's nearest neighbor, whereas this is the case in the Delaunay graph whenever we have found an object p, all of whose nearest neighbors (rather than just k) are farther from q.

There are several reasons for this failure. The first can be seen by examining Figure 4.67(a–c) where for low values of k (i.e., $k \le 3$), the kNN graphs are not connected, as is the case when the kNN graph consists of disconnected subgraphs corresponding to clusters. These clusters have the same effect on the search process as local minima or maxima in optimization problems. This shortcoming can be overcome by increasing k. However, this action does have the effect of increasing the storage requirements of the data structure by a factor of N for each unit increase in the value of k.

Nevertheless, even if we increase k so that the resulting kNN graph is connected, the algorithm may still fail to find the nearest neighbor. In particular, this is the case when the search halts at an object p, that is closer to q than any of p's k nearest neighbors but not necessarily closer to q than any of the objects that are in p's neighbor set but are farther from p than the k nearest neighbors of p. For example, consider the query object X in the kNN graph for $k = 4$ in Figure 4.67(d) positioned two-thirds of the way between k and r so that r is its nearest neighbor. Also consider a search that starts out at any one of objects e, f, j, k, l, m, n. In this case, assuming the planar embedding of Figure 4.64, the search will return k as the nearest neighbor instead of r. Again, this shortcoming can be overcome by further increasing the value of k. However, we do not have a guarantee that the true nearest neighbor will be found.

Sebastian and Kimia [1704] propose to overcome this drawback by extending the search neighborhood around the arbitrary object that is used as the starting point (termed a *seed*) of the search, as well as its closest neighbors obtained through use of the kNN graph, by a factor of $\tau \ge 1$. In particular, an object p_i is said to be τ-*closer* to q with respect to p if $d(p_i,q) \le \tau \cdot d(p,q)$. Armed with this definition, given an arbitrary object p, an element p_n belongs to the extended neighborhood $\text{E}\textsc{n}(p,q)$ of p with respect to query object q if there exists a path $p_0, p_1, \ldots, p_n$, where $p = p_0$ and p_i is τ-closer to q with respect to p for all $i = 1,\ldots,n-1$. An object p is now reported as the nearest neighbor of q if it is closer to q than any other object in $\text{E}\textsc{n}(p,q)$. In essence, use of the extended neighborhood enables us to be able still to get to the nearest neighbor of q when the search is at an object p that is closer to q than any of p's k nearest neighbors but not necessarily closer to q than some of the objects that are reachable by transitioning via one of the k nearest neighbors of p. Note that the quality and performance of the